Automatic Boundary Recognition for Thermal Fluid-Structure Interaction

by Nicklas Linder
March 2009
Supervision: Plamen Pironkov M. Sc.

TECHNISCHE
UNIVERSITÄT
DARMSTADT

fnb Fachgebiet
Numerische Berechnungsverfahren
im Maschinenbau

Abstract

The present Bachelor Thesis deals with the simulation of thermal fluid structure interaction, espacially with the recognition of boundary conditions between the solid and the fluid. The first part introduces the theoretical background of the finite volume and finite element methods, which are needed to solve the fluid flow and the structual analysis numerically. In addition, the k-ϵ method for simulating turbulent flows and the transfinte interpolation for grid generation are presented. Afterwards, this work illustrates how these theories work together in the fluid structure interaction (FSI). When simulating an FSI, one has to consider the coupling technique. In this work, the weak coupling with a partition approach is applied to use the software FEAP[1] for the structural part and FASTEST[2] for the fluid flow. The coupling of these programs is provided by the software MpCCI[3]. Then the newly developed functions for recognising the boundary conditions automatically are introduced and explained. In the last part some complex simulations are presented validating the functionality.

[1] Finite Element Analysis Program - `http://www.ce.berkeley.edu/~rlt/feap/`

[2] Flow Analysis Solving Transport Equations with Simulated Turbulence - `http://www.fnb.tu-darmstadt.de/de/software/fastest/`

[3] Mesh-based parallel Code Coupling Interface - `http://www.mpcci.de`

Statutory declaration

I hereby declare that I wrote this thesis independently and without use of other sources than acknowledged. Passages taken literally or analogously from published or non published sources are marked as such. Drawings or figures in this thesis have either been created by myself or their source is given. This work has not been presented to an examination board in this or a related form previously.

Darmstadt, February 28, 2009

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Nowadays thermal fluid structure analysis play a decisive role in industries and research. It is often not sufficient to consider the structural mechanics, fluid mechanics or the heat transfer individually, as there are significant coupling effects like deformation of a structure caused by thermal expansion and fluid forces or heat transfer due to friction of a fluid. If one wants to simulate these phenomena, one has to decide between different coupling approaches. On the one hand, one can use one solver for both, the solid and fluid (strong coupling) computing the problems simultaneously or the domains can be split up into two solvers with an coupling interface, that provides the exchange of the forces, deformations and temperatures after each timestep (weak coupling). Applying the latter, one can use the software MpCCI[1] in order to couple the software FEAP[2] for the structure mechanics analysis and FASTEST[3] for the fluid mechanics.

1.1 Motivation

When using MpCCI as coupling interface for FEAP and FASTEST, the user has to provide a definition of the coupling surface. The coupling succeeds, when MpCCI gets from both softwares a certain amount of nodes defining the same coupling surface. This definition is already implemented for FASTEST by the use of a grid generator like *icemCFD*. In FEAP the definitions have to be done manually in the input file. Each coupling node and element have to be defined explicitly. For more complex geometries this leads to extremely large input files, which are hard to be parametrized what implicates, that the definition has to be updated when the mesh changes due to refining or editing. It is obvious that one can not handle the definitions manually for complex geometries and the need for an automatisation gets clear. This is what was done in this work. The developed functions provide a recognition of coupling nodes and elements by the use of parameters that define this surface uniquely, e.g. three points for a planar coupling face.

With the use of the new functions, simulation of fluid-structure interactions using FEAP and FASTEST gets more interesting as the user can also simulate more complicate geometries or edit the mesh whenever needed.

1.2 Objective and outline

The objection of this theses is to identify the coupling nodes for different geometries (planar, polar, coned and freeform) only with a bunch of parameters which define the surface exactly. This includes the implementation of *usermacros* that provide the user interaction in the FEAP input file to pass the parameters. These parameters have to be conditioned for the use of computations within the recognition-routines. Founded nodes then have to be prepared to be sent to MpCCI in respect to the requirements the software has.

To complete this project, the implemented functions have to be evaluated, documented and also tested in more complex geometries.

[1] Mesh-based parallel Code Coupling Interface - `http://www.mpcci.de`

[2] Finite Element Analysis Program - `http://www.ce.berkeley.edu/~rlt/feap/`

[3] Flow Analysis Solving Transport Equations with Simulated Turbulence - `http://www.fnb.tu-darmstadt.de/de/software/fastest/`

2 Fundamentals

For the numerical solution of problems in fluid and continuum mechanics, the *finite volume method* for the fluid and the *finite element method* for the structural problems are mainly employed. They are based on mathematical concepts, definitions and methods which will be discussed in the following chapters. Afterwards the *k-ϵ method* is introduced for the computation of turbulent flows. Especially in context of coupling free form surfaces, the grid generation via *transfinite interpolation* is an important tool and will be described at the end.

2.1 Governing equations for fluid mechanical problems

In order to characterize the flow behaviour of liquids or gasses (possibly with additional consideration of heat and species transport processes) one usually employs the Eulerian formulation (a spaciously fixed reference system). This formulation allows to describe the properties of a flow at a certain location in the flow domain.

The *mass m* of an arbitrary volume V is defined by

$$m(t) = \int_V \rho(x,t)\, dV \tag{2.1}$$

with the *density* ρ. If there are no mass sources or sinks, the total mass of a body remains constant for all time:

$$\frac{D}{Dt}\int_V \rho\, dV = 0. \tag{2.2}$$

During a deformation, the volume and the density may change, but the mass will be the same. Written in differential formulation yields to

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho v_i)}{\partial x_i} = 0. \tag{2.3}$$

We restrict ourselves to the *Newtonian fluids* which are characterized by the following material law for the Cauchy stress Tensor **T**:

$$T_{ij} = \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3}\frac{\partial v_k}{\partial x_k}\delta_{ij}\right) - p\delta_{ij} \tag{2.4}$$

with the *pressure p* and the *dynamic viscosity* μ. Using this tensor together with the conservation laws for mass, momentum and energy and again applying Fourier's law yields to

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho v_i)}{\partial x_i} = 0 \tag{2.5}$$

$$\frac{\partial(\rho v_i)}{\partial t} + \frac{\partial(\rho v_i v_j)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3}\frac{\partial v_k}{\partial x_k}\delta_{ij}\right)\right] - \frac{\partial p}{\partial x_i} + \rho f_i \tag{2.6}$$

$$\frac{\partial(\rho e)}{\partial t} + \frac{\partial(\rho v_i e)}{\partial x_i} = \mu\left[\frac{\partial v_i}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) - \frac{2}{3}\left(\frac{\partial v_k}{\partial x_k}\right)^2\right] - p\frac{\partial v_i}{\partial x_i} + \frac{\partial}{\partial x_i}\left(\kappa\frac{\partial T}{\partial x_i}\right) + \rho q. \tag{2.7}$$

(2.6) is known as the *(compressible) Navier-Stokes equation*. Two more equations are needed, as we have more unknowns than equations. They can be derived from the thermodynamic properties of the fluid, namely the *thermal* and the *caloric equations of states*. If we consider *caloric ideal gas*, they are

$$p = \rho R T \quad \text{and} \quad e = c_v T \tag{2.8}$$

with the *specific gas constant R* of the fluid and the *specific heat capacity* c_v. These equations can be simplified by assuming incompressible an inviscid flows e.g.. Together with appropriate boundary conditions we now have a closed formulation of a fluid mechanical problem.

(2.6) can be simplified for incompressible flows where $\partial v_i/\partial x_i = 0$ which leads to

$$\frac{\partial v_i}{\partial x_i} = 0, \tag{2.9}$$

$$\frac{\partial (\rho v_i)}{\partial t} + \frac{\partial \left(\rho v_i v_j\right)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\right] - \frac{\partial p}{\partial x_i} + \rho f_i, \tag{2.10}$$

$$\frac{\partial (\rho e)}{\partial t} + \frac{\partial (\rho v_i e)}{\partial x_i} = \mu \frac{\partial v_i}{\partial x_j}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) + \frac{\partial}{\partial x_i}\left(\kappa \frac{\partial T}{\partial x_i}\right) + \rho q. \tag{2.11}$$

For a criterion for the validility of this assumption the *Mach number*

$$Ma = \frac{\bar{v}}{a} \tag{2.12}$$

is taken into account, where $\bar{v}$ is a characteristic flow velocity of the problem and a is the *speed of sound* in the corresponding fluid. Incompressibility usually is assumed if $Ma < 0.3$.

How these equations can be solved will be presented in (2.4)

2.2 Governing equations for continuum mechanics problems

When dealing with structural mechanics, the *Lagrangian formulation* is preferred, where the reference system is linked to a material point of the solid described by the transformation

$$x_i = \chi(X_j, t). \tag{2.13}$$

The *momentum conservation law* states, that the temporal change of the momentum of a body equals the sum of all body and surface forces acting on the body. The *momentum vector* $\mathbf{p} = p_i e_i$ of a body is defined by:

$$p_i(t) = \int_V \rho(x,t) v_i(x,t) dV \tag{2.14}$$

(with the density $\rho(x,t)$ and the velocity $v(x,t)$). The differential formulation of this law yields

$$\frac{\partial(\rho v_i)}{\partial t} + \frac{\partial(\rho v_i v_j)}{\partial x_j} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i. \tag{2.15}$$

In this context, f_i denotes the acting volume forces and T_{ij} are the components of the symmetric *Cauchy stress tensor* which describes the body's state of stress.

For linear problems, where only small deformations are allowed, the strain of each point is characterized by the strain displacement relation

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \tag{2.16}$$

(2.15) formulated for displacements leads to

$$\rho \frac{D^2 u_i}{Dt^2} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i. \tag{2.17}$$

For the unknown tensor T_{ij} we have to assume a material law like *Hooke's law* for linear elastic material behaviour:

$$T_{ij} = \lambda \epsilon_{kk} \delta_{ij} + 2\mu \epsilon_{ij}. \tag{2.18}$$

λ and μ are the *Lamé constants*, which depend on the corresponding material and can be derived from the *elasticity modulus E* and the *Poisson ratio* ν:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad \text{and} \quad \mu = \frac{E}{2(1+\nu)}. \tag{2.19}$$

Now (2.17) combined with (2.18) yields to the *Navier-Cauchy equations of linear elastic theory*:

$$\rho \frac{D^2 u_i}{Dt^2} = (\lambda + \mu)\frac{\partial^2 u_j}{\partial x_i \partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + \rho f_i. \tag{2.20}$$

In association with appropriate boundary and initial conditions (e.g. prescribed displacements and/or stresses), (2.20) denotes a closed system of partial differential equations which can be used for the determination of the unknown displacements u_i. For non-linear material properties or great body deformations theses equations are not suitable but this is not within the scope of this work. Additional information can be found in [11] and [7].

2.3 Governing equations for heat transfer problems

For simple cases like diffusion in solids or diffusion and convection in fluids the heat transfer can be described by the *scalar transport equations*. As the heat conduction in solids is a special case of heat transfer in fluids, the latter will be discussed here.

The *first law of thermodynamics* states that *the total energy W* of a body equals the total external energy supply which is composed by *the power of external forces* P_a and *the heat supply* Q:

$$\frac{DW}{Dt} = P_a + Q. \tag{2.21}$$

The *total energy W* of a body is defined by

$$W(t) = \int_V \rho e dV + \frac{1}{2}\int_V \rho v_i v_i dV \tag{2.22}$$

with e the *specific internal energy*. P_a and Q are defined by

$$P_a(t) = \int_S T_{ij} v_j n_i dS + \int_V \rho f_i v_i dV \quad \text{and} \quad Q(t) = \int_V \rho q dV - \int_S h_i n_i dS \tag{2.23}$$

where q denotes (scalar) heat sources and $\mathbf{h} = h_i e_i$. With these definitions, (2.21) can be written as:

$$\frac{D}{Dt}\int_V \rho\left(e + \frac{1}{2}v_i v_i\right) dV = \int_S \left(T_{ij} v_j - h_i\right) n_i dS + \int_V \rho\left(f_i v_i + q\right) dV \tag{2.24}$$

with the (known) velocity $v = v_i e_i$. Using Gauss integral theorem, the momentum conservation law 2.15 and the *Reynolds transport theorem*, one obtains the energy balance in the differential form:

$$\frac{\partial\left(\rho e\right)}{\partial t} + \frac{\rho\left(\rho v_i e\right)}{\partial x_i} = T_{ij}\frac{\partial v_j}{\partial x_i} - \frac{\partial h_i}{\partial x_i} + \rho g \tag{2.25}$$

Now we assume *Fourier's Law* (for isotropic materials)

$$h_i = -\kappa\frac{\partial T}{\partial x_i} \tag{2.26}$$

(where κ is the *heat conductivity*) as a constitutive relation for the heat flux and a flow. In addition we assume, that κ of the fluid is constant and the work done by pressure and friction forces can be neglected. Then the energy balance 2.25 yields to the convection-diffusion equation for the temperature T

$$\partial\frac{\left(\rho c_p T\right)}{\partial t} + \frac{\partial}{\partial x_i}\left(\rho c_p v_i T - \kappa\frac{\partial T}{\partial x_i}\right) = \rho q, \tag{2.27}$$

where q donates possibly present heat sources or sinks and c_p the specific heat capacity.

In order to solve this equation, one has to adopt certain boundary conditions. These can be

- prescribed temperatures: $T = T_b$
- prescribed heat flux: $\kappa\frac{\partial T}{\partial x_i}n_i = h_b$
- heat flux proportional to heat transport: $\kappa\frac{\partial T}{\partial x_i}n_i = \tilde{\alpha}\left(T_b - T\right)$

where T_b and h_b are prescribed values at the problem domain boundary Γ for the temperature and the heat flux in normal direction, respectively, and $\tilde{\alpha}$ is the *heat transfer coefficient*.

If one now set $v_i = 0$, one considers only the diffusion and obtains the heat conduction equation in a medium at rest (fluid or solid). If one now also drops the terms with the time derivation, the corresponding equations for steady heat transfer are obtained:

$$\kappa\frac{\partial^2 T}{\partial x_i^2} = \rho q \tag{2.28}$$

2.3.1 Linear Thermo Elasticity

Quite often, thermal effects play an essential role for structural deformation. Therefore it is necessary to find equations describing a coupled thermo-mechanical problem. They can be derived from the momentum law (2.15) and the energy conservation (2.24) using the strain displacement relation (2.16). We assume a simple linear thermo-elastic material.

The *specific dissipation function* ψ is given as:

$$\psi = T_{ij}\frac{D\epsilon_{ij}}{Dt} - \rho\frac{D}{Dt}(e - Ts) + \rho s\frac{DT}{Dt} \tag{2.29}$$

where s is the *specific internal entropy*. This function is a measure for the energy dissipation in the continuum. As we assume a simple thermo elastic material, there is no energy dissipation. Therefore $\psi = 0$. Using this equation together with the energy conservation (2.24) and again assuming the validity of Fourier's Law (2.26) yields to

$$T\rho\frac{Ds}{Dt} = -\frac{\partial}{\partial x_i}\left(\kappa\frac{\partial T}{\partial x_i}\right) + \rho q \tag{2.30}$$

2.4 Finite Volume Method

As there does not exist an universal algebraic solution for the partial differential equation introduced before, the solutions have to be approximated. Different methods have been developed and the most frequently used for fluid flow problems is the *finite volume method* which will be introduced here for the two dimensional case. The equations for the three dimensional case follow straight forward. For mechanical problems, the *finite element method* will be outlined in the next chapter.

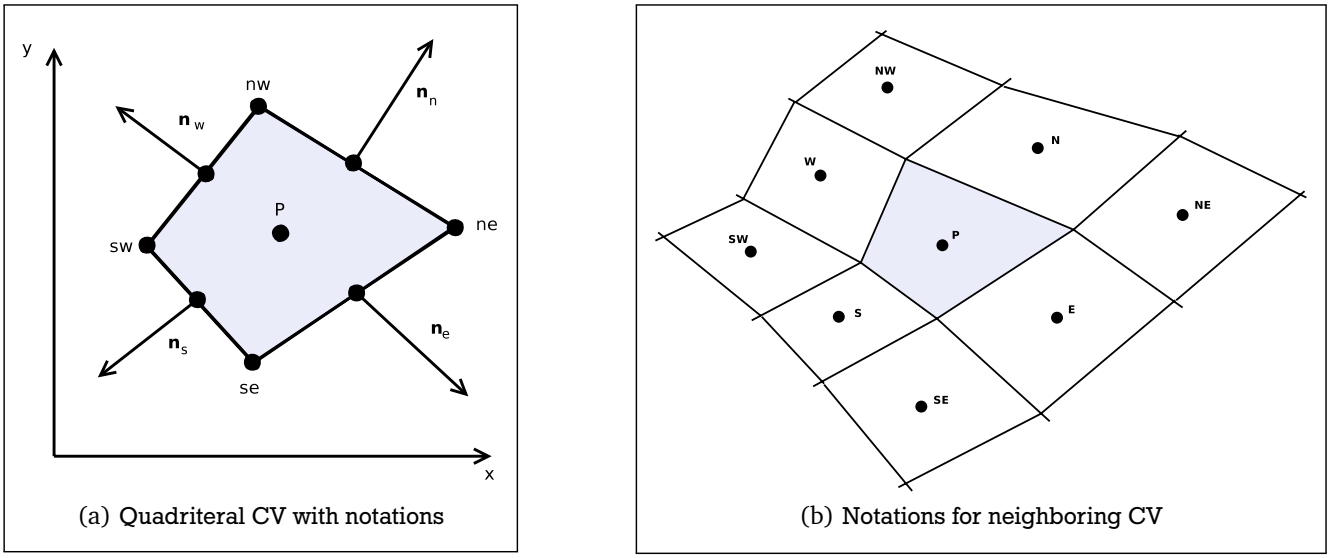
2.4.1 Discretisation

As a first step, the problem domain Ω will be discritized into a finite number of non overlapping subdomains V_i $(i = 1, ..., N)$, the *control volumes (CV)* and related nodes, where the variable values are computed. They can be defined on a numerical grid, which can be generated with different techniques (details in [12], [6] and [7]). If one now integrates the differential balance equations over an arbitrary control volume V and applies the Gauss integral theorem, one obtains for the general stationary transport equation (2.28)

$$\int_S \left(\rho v_i\phi - \alpha\frac{\partial\phi}{\partial x_i}\right) n_i dS = \int_V f\,dV, \tag{2.31}$$

where S is the surface of the CV and n_i are the components of the unit normal vector of the surface.

The same notation as in [6] is used here: in compass notation the four sides and the four vertexes are named. The neighbours of the element are also named in compass notation but with capitol letters (see image 2.1)



(a) Quadriteral CV with notations

(b) Notations for neighboring CV

Figure 2.1: Notation if the CV

Then the surface integral (2.31) can be split into the sum of the four surface integrals over the faces S_c (c = e, w, n, s) of the CV:

$$\sum_c \int_{S_c} \left(\rho v_i \phi - \alpha \frac{\partial \phi}{\partial x_i} n_{ci} \right) dS_c = \int_V f \, dV. \tag{2.32}$$

Physically this can be interpreted as the sum of the convective and diffusive fluxes F_c^C and F_c^D through the CV faces, respectively.

Looking at the face S_e, for instance, the unit normal vector $n_e = (n_{e1}, n_{e2})$ is defined by

$$\mathbf{n}_e = \frac{(y_{ne} - y_{se})}{\delta S_e} \mathbf{e}_1 - \frac{(x_{ne} - x_{se})}{\delta S_e} \mathbf{e}_2 \tag{2.33}$$

where

$$\delta S_e = |\mathbf{x}_{ne} - \mathbf{x}_{se}| = \sqrt{(x_{ne} - x_{se})^2 + (y_{ne} - y_{se})^2} \tag{2.34}$$

denotes the length of the face S_e. Until here, no approximation has been made, and the equations are still exact.

2.4.2 Approximation of Surface and Volume Integrals

Now the integrals in (2.32) have to be approximated as there is no direct solution. We only consider the two dimensional case here, but the three dimensional can be adopted straight forward. Several methods for the approximation of the surface integral can be used like the *midpoint rule*, *trapezoidal rule* and the *Simpson rule*.

Let us have a look at the approximation of a general surface integral

$$\int_{S_e} w_i n_{ei} dS_e \tag{2.35}$$

over the face S_e of a CV. Other faces can be treated in an analogous way. w_i is a general integrand function $w = (w_1(x), w_2(x))$. Applying the *midpoint rule* for this integral means to use the value of the midpoint of the CV as an approximation of the integral. Other approximation rules can be found in [6] and will not be discussed here.

Using the midpoint rule for the convective and diffusive fluxes through the CV faces in 2.32 we obtain

$$F_c^C \approx \underbrace{\rho v_i n_{ci} \delta S_c}_{\dot{m}_c} \phi_c \quad \text{and} \quad F_c^D \approx -\alpha n_{ci} \delta S_c \left(\frac{\partial \phi}{\partial x_i} \right)_c \tag{2.36}$$

if v_i, ρ and α are constant across the CV, what we assume here. With the use of the definition of the normal vector, the convective flux through the face S_c can be approximated.

Considering the volume integral in (2.32) the approximation can be done with the assumption that the value of the centre f_p of the CV represents an average value over the CV. This leads to to the two dimensional midpoint rule:

$$\int_V f \, dV \approx f_p \delta V, \tag{2.37}$$

where δV denotes the volume of the CV. There are several integration formulas for Cartesian grids that differs in the corresponding error order (with respect to δV). They can be found in [6] and will not be described here. The integration formulas for three dimensional volume integrals are available analogously.

The midpoint rule applied to all integrals leads to the following approximated balance equation of (2.32):

$$\sum_c \dot{m}_c \phi_c - \sum_c \alpha n_{ci} \delta S_c \left(\frac{\partial \phi}{\partial x_i} \right)_c = f_p \delta V. \tag{2.38}$$

Now the function values and derivatives of ϕ at the CV which occur in the convective and diffusive fluxes have to be approximated by the variable values in the CV centre. This will be outlined for two dimensional Cartesian grid and can be transferred as well to non Cartesian and three dimensional grid.

Discretisation of Convective Fluxes

What should be done next, is to approximate the values of ϕ_c in the neighbourring CV. Therefore exist several different techniques. The *central differencing scheme (CDS)* and the *upwind differencing scheme (UDS)* are two of them and they will be explained. Other techniques like the *flux blending technique* are described in [6].

The CDS approximates ϕ_e by the use of the centre point P of the CV and the centre point of the eastern neighbour E:

$$\phi_e \approx \gamma_e \phi_E + (1 - \gamma_e)\phi_P. \tag{2.39}$$

The interpolation factor γ_e is defined by

$$\gamma_e = \frac{x_e - x_P}{x_E - x_P}. \tag{2.40}$$

For Cartesian grids, this approximation has for equidistant and non equidistant grids an interpolation error of second order and this can be proved by the use of a Taylor series expansion of ϕ around the point x_p. Approximation of a higher order can be found by involving additional grid points. Here one should also use an integration formula of corresponding order.

The UDS determines ϕ_e depending on the direction of the mass flux as follows

$$\phi_e = \phi_P, \quad if \quad \dot{m}_e > 0, \tag{2.41}$$
$$\phi_e = \phi_E, \quad if \quad \dot{m}_e < 0 \tag{2.42}$$

and has an interpolation error of first order (this can be shown by a Taylor series expansion of ϕ around x_P, evaluated at the point x_e).

This approximation is quite good, if the transport direction is nearly perpendicular to the CV face. Otherwise the approximations can be quite inaccurate and for large mass fluxes it can be necessary to employ very fine grids for the computation.

Discretisation of Diffusive Fluxes

To approximate the diffusive fluxes, it is necessary to approximate the values of the normal derivative of ϕ at the CV faces by nodal values in the CV centres.

Assuming, that ϕ is a linear function between the points x_P and x_E, one obtains a central differential formula:

$$\left(\frac{\partial \phi}{\partial x}\right)_e \approx \frac{\phi_E - \phi_P}{x_E - x_P}. \tag{2.43}$$

The error of the approximation is of second order for equidistant grid and again can be proved by the use of Taylor expansion around x_e at the locations x_P and x_E. For non-equidistant grid, the approximation error is proportional to the grid spacing.

Other techniques of higher approximation order can be found in [6] or [7] as well as the necessary modifications for non Cartesian grids which will not be discussed here.

2.4.3 Discrete Transport Equation

Employing the midpoint rule for the integral approximations, the UDS method for the convective flux, the CDS method for the diffusive flux and assume that the velocity components $v_1, v_2 > 0$ on a Cartesian grid yields to a relation of the form

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + b_P \tag{2.44}$$

with the coefficients

$$a_E = \frac{\alpha}{(x_E - x_P)(x_e - x_W)}, \tag{2.45}$$

$$a_W = \frac{\rho v_1}{x_e} + \frac{\alpha}{(x_P - x_W)(x_e - x_W)}, \tag{2.46}$$

$$a_N = \frac{\alpha}{(y_N - y_P)(y_n - y_s)}, \tag{2.47}$$

$$a_S = \frac{\rho v_2}{y_n y_s} + \frac{\alpha}{(y_P - y_S)(y_n - y_s)}, \tag{2.48}$$

$$a_P = \frac{\rho v_1}{x_e x_w} + \frac{\alpha (x_E - x_W)}{(x_P - x_W)(x_E - x_P)(x_e - x_w)} + \frac{\rho v_2}{y_n y_s} + \frac{\alpha (y_N - y_S)}{(y_P - y_S)(y_N - y_P)(y_n - y_s)}, \tag{2.49}$$

$$b_P = f_p. \tag{2.50}$$

The relation of the coefficients

$$a_P = a_E + a_W + a_N + a_S \tag{2.51}$$

is characteristic for the finite volume discretisation and expresses the conservativity of the method.

2.4.4 Treatment of Boundary Conditions

When dealing with boundary value problems, boundary conditions are needed to solve them numerically. There are three boundary condition types that most frequently occur for the considered type of problems:

- a prescribed variable value
- a prescribed flux
- and a symmetry boundary

If we have a prescribed boundary value $\phi_w = \phi^0$, the convective flux at the boundary yields to:

$$F_W^C \approx \dot{m}_w \phi_w = \dot{m}_w \phi^0 \tag{2.52}$$

Now F_W^C, as well as $\dot{m}_w$, are known (see above) and can be used in the balance equation (2.38).
With the same approach as in the interior of the domain, the diffusive flux through the boundary is determined:

$$\left(\frac{\partial \phi}{\partial x}\right)_W \approx \frac{\phi_P - \phi_W}{x_P - x_W} = \frac{\phi_P - \phi^O}{x_P - x_W} \tag{2.53}$$

If we now consider a prescribed flux at a boundary (e.g. $F_W = F^0$) the flux through the CV face is obtained by dividing F^0 through the length of the face (e.g. $x_e - x_W$). The resulting value is introduced in (2.38) as total flux.
If a problem can be split into two or more symmetric domains, one can downsize the problem domain to save computing time or to get a higher accuracy (with a finer grid) with the same computational effort. Boundary conditions of this type can be applied using:

$$\frac{\partial \phi}{\partial x_i} n_i = 0 \tag{2.54}$$

which means that the diffusive flux through the symmetry boundary is zero. At a symmetry boundary, the normal component of the velocity vector has to be zero. Therefore the mass flux is zero which causes that the convective flux through the boundary is also zero. Thus the total flux through the corresponding CV face can be set to zero in the balance equation.
Together with the boundary conditions at all boundaries of the problem domain the algebraic system of equations resulting from a finite volume discretisation has an unique solution which is described in the next chapter.

2.4.5 Algebraic System of Equation

We now have an algebraic equation for each CV. Summed up over all N CVs

$$a_P^i \phi_P^i - \sum_c a_c^i \phi_c^i = b_P^i \tag{2.55}$$

with $(c = n, e, s, w)$. In the one dimensional case, the centre point of a CV has only one eastern and one western neighbour. Therefore the equation for the $i-th$ CV with a second order central differencing scheme is

$$a_P^i \phi_P^i - a_E^i \phi_E^i - a_W^i \phi_W^i = b_P^i \tag{2.56}$$

whereas

$$\phi_W^i = \phi_P^{i-1} \quad \text{for all} \quad i = 2, ..., N \quad \text{and} \tag{2.57}$$
$$\phi_E^i = \phi_P^{i+1} \quad \text{for all} \quad i = 1, ..., N-1. \tag{2.58}$$

For 2D it is analogously. Written in matrix form this yields to

$$\mathbf{A}^{N \times M} \phi^{1 \times N} = b^{1 \times N}. \tag{2.59}$$

whereas **A** has a diagonal form.

The solution of the equation will be explained in Chapter 2.6 as it is also used for the finite element method which is introduced in the next chapter.

2.5 Finite Element Method

The following chapter introduces the *Finite Element Method (FEM)* which is most frequently used for an approximation of the equations governed in 2.2.

2.5.1 Discretisation

Analogously to the FVM, the problem domain has to be discretized before the mathematical foundations of the FEM can be considered. This is done by the use of a finite number of non overlapping elements that might be triangles or quadrilateral structures in a two dimensional case.

The state of an element is usually described by the piecewise polynomial *ansatz functions*. These are formulated by the use of designated local attributes Φ_1^i,..,Φ_P^i such as node values and/or derivatives (of the i-th element) in contrast to the FVM where these are values of the element centre. The approximation can be written as

$$\Phi^i(x) = \sum_{j=1}^{p} \Phi_j^i N_j^i(x) \tag{2.60}$$

with the *local shape functions* N_1,...,N_p and the unknown Φ_j^i. If only function values are used as nodal variables, i.e. at suitable locations $x_1, \ldots, x_p$ in the Element E_i, the local shape functions fulfill the relations

$$N_j^i(x_n) = \begin{cases} 1, & \text{for} \quad j = n \\ 0, & \text{for} \quad j \neq n \end{cases} \tag{2.61}$$

since Φ^i at the nodes x_n must take the nodal value Φ_n^i.

A global representation can be derived when numbering the problem domain consecutively, where commom local nodal variables of adjoint elements are counted only once:

$$\Phi(x) \approx \phi_0(x) + \sum_{k=1}^{N} \Phi_k N_k(x) \tag{2.62}$$

N_k is that local shape function, where the local nodal variable Φ_j^i coincides with the global nodal variable Φ_k. In the function ϕ_0 the Dirichlet boundary conditions are subsummed.

Neumann boundary conditions

$$\frac{\partial \phi}{\partial x_i} n_i = t_b \tag{2.63}$$

can be adopted in the *load vector* b (see 2.5.3).

2.5.2 Method of the Weighted Residuals

As we have introduced an approximation for the solution of the equations, the *residual* R is introduced, which represents the current deviation from the exact solution. We now claim that the integral mean value of R over the problem domain vanishes. This leads to conditions for the unknown values Φ_1^i ,...,Φ_p^i. To determine the mean value one can use an arbitrary *test function* ϕ which vanishes on all *Dirichlet* boundaries (prescribed value at a certain point) of the problem domain Ω. This approach is called the *method of weighted residuals* and can be written as

$$\int_\Omega R\phi \, d\Omega \stackrel{!}{=} 0 \tag{2.64}$$

2.5.3 The Galerkin Method and the Resulting System of Equation

Using the ansatz functions from (2.5.1) as test functions as well, (2.64) leads to the formulation of the *Galerkin method*. In combination with an adequate numerical integration scheme, all local formulations might be incorporated into a (preliminary) global linear system of equations for the unknown values Φ_1^i ,...,Φ_p^i for each timestep.

To solve the system of equations, boundary conditions are necessary. They can be applied straight forward as prescribed displacements directly determine the values of the corresponding unknowns. Prescribed stresses can be integrated into the right side of the resulting system

$$A\Phi = b \tag{2.65}$$

with the sparsely filled *stiffness matrix* A, the *load vector* b and the unknowns Φ.

2.6 Solution of Linear Systems of Equations

Both, the FVM and the FEM have a linear system of equations to solve. Both are of the form $A\Phi = b$, whereas A is a sparsely filled matrix. In the recent years, many powerful methods for efficient solving of the equations have been developed. Several direct solvers like the *Gaussian elimination* compute the exact solution with at most n^3 operations, iterative solvers compute an approximation of the solution and usually have a linear complexity. These are solvers like the *Jacobi and Gauss-Seidel* that will be shortly introduced here. More solvers and theory can be found in [6].

2.6.1 Jacobi and Gauss-Seidel

Both method requires, as a first step, a decomposition of the system matrix A similar to

$$A = L + D + U \tag{2.66}$$

with the lower triangle L, the upper triangle U and the diagonal part D of A. The algorithms differ in the formulation of the iteration step. For the Jacobi method it may be written as

$$D\dot{\Phi}^{k+1} = -(L+U)\dot{\Phi}^k + b. \tag{2.67}$$

whereas the Gauss-Seidel uses

$$(D+L)\Phi^{k+1} = -U\Phi^k + b. \tag{2.68}$$

Although the system has to be solved for each iteration step, it is mostly more efficient than inverting the sparsely filled matrix A. Further iterative methods can be found in [1].

2.7 Computation of Turbulent Flows

In practical applications, most flow processes are turbulent. The introduced Navier-Stokes equations in Sec. (2.1) could theoretically be taken into account for simulating the flow, but practically it is not possible due to the enormous computational effort. Therefore special numerical methods were introduced to handle the complexity. They are mostly based on statistical turbulence models and we will restrict ourselves to the incompressible case.

2.7.1 Characterization of Computational Methods

A decisive factor for the simulation of turbulent flows are the great scales of length and time dimensions and consequently the high resolution. They have been characterised by Kolmogorov (1942) and are directly coupled to the Reynold's number (a dimensionless number that gives a measure of the ratio of inertial forces to viscous forces). The spatial scale must be $l_k \sim Re^{3/4}$ and the time scale $t_k \sim Re^{1/2}$ in order to fully compute the flow. Because of these requirements the *direct numerical simulation (DNS)* is not practically usable for great Reynoldsnumbers due to the enormous computational effort of the great amount of timesteps and gridpoints.

Therefore alternative approaches have been employed for computing turbulent flows and will be introduced in the next chapter. We restrict ourselves to the *statistical turbulence models*. Other approaches like the *large eddy simulation (LES)* can be found in [5] and [6].

2.7.2 Statistical Turbulence Modelling

When using a statistical turbulence model, each flow variable Φ is expressed by a mean value $\bar{\phi}$ and a fluctuation ϕ':

$$\phi(\mathbf{x}, t) = \bar{\phi}(\mathbf{x}) + \phi'(\mathbf{x}, t), \tag{2.69}$$

with the mean value

$$\bar{\phi}(\mathbf{x}) = \lim_{T\to\infty} \frac{1}{T} \int_{t_o}^{t_0+T} \phi(\mathbf{x}, t) dt \tag{2.70}$$

in the *statistically steady* case. If the averaging time T is large enough, $\bar{\phi}$ does not depend on the point of time t_0 when the averaging started. If ϕ is statistically unsteady, the mean value is time dependent too. It is then defined by *ensemble averaging*:

$$\bar{\phi}(\mathbf{x}, t) = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \phi(\mathbf{x}, t). \tag{2.71}$$

Inserting this ansatz in the conservation equations for mass, momentum and energy (2.9) and subsequently averaging yields to the *Reynolds averaged Navier-Stokes (RANS) equations* (or *Reynolds equations)*:

$$\frac{\partial \bar{v}_i}{\partial x_i} = 0, \tag{2.72}$$

$$\frac{\partial(\rho \bar{v}_i)}{\partial t} + \frac{\partial}{\partial x_j}\left[\rho \bar{v}_i \bar{v}_j + \overline{v_i' v_j'} - \mu\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right)\right] + \frac{\partial \bar{p}}{\partial x_i} = p f_i \tag{2.73}$$

We now have simplified equations as the mean values are time independent or at least the time dependence can be resolved with a certain amount of time steps. But we have as well the *Reynolds stresses* $\rho \overline{v_i' v_j'}$ as new unknowns. To solve the equations system, suitable approximations for the correlations have to be employed. There are different models for this *turbulence modelling* whereas the most important are:

- algebraic models (zero-equation models),
- one- and two-equation models,
- Reynold stress models.

In this work the *k-ϵ model* was used.

2.7.3 The k-ϵ Turbulence Model

The k-ϵ model, developed in the 1960s by Spalding and Launder, assumes that

$$\rho\overline{v_i'v_j'} = -\mu_t\left(\frac{\partial\bar{v}_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i}\right)+\frac{2}{3}\rho\delta_{ij}k \tag{2.74}$$

for the Reynolds stresses (also known as *Bussinesq approximation*). μ_t denotes the *turbulent viscosity* and depends on the flow variables, δ_{ij} is the *Kronecker function* and k is the *turbulent kinetic energy*, defined by

$$k = \frac{1}{2}\overline{v_i'v_i'}. \tag{2.75}$$

Since μ_t and k are unknown too, the system is not yet well defined. Two other assumptions of the k-ϵ model relates μ_t to k and to the *dissipation rate* of the turbulent kinetic energy ϵ:

$$\mu_t = C_\mu\rho\frac{k^2}{\epsilon} \quad , \quad \epsilon = \frac{\mu}{\rho}\overline{\frac{\partial v_i'}{\partial x_j}\frac{\partial v_i'}{\partial x_j}}. \tag{2.76}$$

Two transport equations for k and ϵ are needed. Inserting 2.74 into the momentum equation (2.5) and defining $\tilde{p} = \bar{p} + 2k/3$ yields to the equation system

$$\frac{\partial\bar{v}_i}{\partial x_i} = 0, \tag{2.77}$$

$$\frac{\partial(\rho\bar{v}_i)}{\partial t}+\frac{\partial}{\partial x_i}\left[\rho\bar{v}_i\bar{v}_j-(\mu+\mu_t)\left(\frac{\partial\bar{v}_i}{\partial\bar{x}_j}+\frac{\partial\bar{v}_j}{\partial\bar{x}_i}\right)\right] = -\frac{\partial\tilde{p}}{\partial x_i}+\rho f_i, \tag{2.78}$$

$$\frac{\partial(\rho k)}{\partial t}+\frac{\partial}{\partial x_j}\left[\rho\bar{u}_j k-(\mu+\frac{\mu_t}{\sigma_k})\frac{\partial k}{\partial\bar{x}_j}\right] = G-\rho\epsilon, \tag{2.79}$$

$$\frac{\partial(\rho\epsilon)}{\partial t}+\frac{\partial}{\partial x_j}\left[\rho\bar{u}_j\epsilon-(\mu+\frac{\mu_t}{\sigma_\epsilon})\frac{\partial\epsilon}{\partial\bar{x}_j}\right] = C_{\epsilon1}G\frac{\epsilon}{k}-C_{\epsilon2}\rho\frac{\epsilon^2}{k}, \tag{2.80}$$

$$\tag{2.81}$$

which has to be solved for the unknowns $\tilde{p}, \bar{v}_i, k$ and ϵ, $C_{\epsilon i}$ are model constants. With this equation system and reasonable boundary conditions one has reduced the computational effort enormously and one can now simulate turbulent flows with an adequate accuracy.

2.8 Transfinite Interpolation

The *transfinite interpolation (TFI)* is usually used for grid generation. The idea is to map a structured mesh from a logical domain to a physical domain. This is illustrated in fig. 2.2.

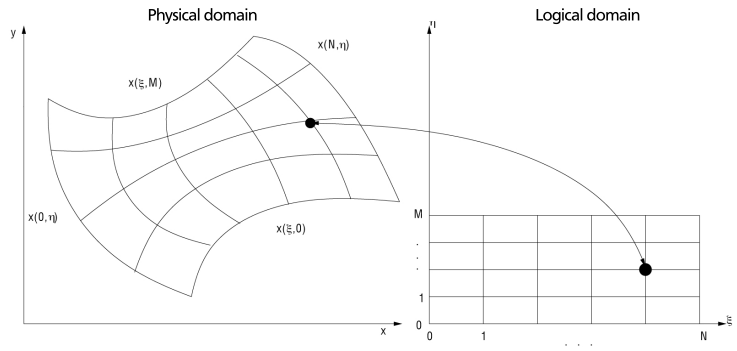


Figure 2.2: Relation between coordinates and grid points in physical and logical domains.

It is now necessary to have unique mapping

$$(x,y)=(x(\xi,\eta),y(\xi,\eta)) \quad \text{or} \quad (\xi,\eta)=(\xi(x,y),\eta(x,y)) \tag{2.82}$$

between given discreet values $\xi = 0,1,...,N$ and $\eta = 0,1,...,M$ and the physical problem values.
First, the boundary of the problem domain is described:

$$x(\xi,o)=x_s(\xi), \quad x(\xi,M)=x_n(\xi) \quad \text{for} \quad \xi=0,...,N, \tag{2.83}$$
$$x(0,\eta)=x_w(\eta), \quad x(N,\eta)=x_e(\eta) \quad \text{for} \quad \eta=0,...,M. \tag{2.84}$$

The corner points have to fulfil the compatibility conditions. Now the inner points can be determined by the use of an interpolation rule. A simple linear interpolation yields to the following relation

$$\begin{aligned}\mathbf{x}(\xi,\eta) =&(1-\frac{\eta}{M})\mathbf{x}_s(\xi)+\frac{\eta}{M}\mathbf{x}_n(\xi)+(1-\frac{\xi}{N})\mathbf{x}_w(\eta)+\frac{\xi}{N}\mathbf{x}_e(\eta)\\ &-\frac{\xi}{N}\left[\frac{\eta}{M}\mathbf{x}_n(M)+(1-\frac{\eta}{M})\mathbf{x}_s(M)\right]\\ &-(1-\frac{\xi}{N})\left[\frac{\eta}{M}\mathbf{x}_n(0)+(1-\frac{\eta}{M})\mathbf{x}_s(0)\right]\end{aligned} \tag{2.85}$$

which is known as the TFI. One can get the three dimensional case straight forward. We assume $0 \leqslant \xi \leqslant 1$, $0 \leqslant \eta \leqslant 1$ and $0 \leqslant \zeta \leqslant 1$:

$$\begin{aligned}t_{uu}(\xi,\eta,\zeta) =&(1-\xi)\mathbf{x}(0,\eta,\zeta)+\xi\mathbf{x}(1,\eta,\zeta)\\ t_{vv}(\xi,\eta,\zeta) =&(1-\eta)\mathbf{x}(\xi,0,\zeta)+\xi\mathbf{x}(\xi,1,\zeta)\\ t_{ww}(\xi,\eta,\zeta) =&(1-\zeta)\mathbf{x}(\xi,\eta,0)+\xi\mathbf{x}(\xi,\eta,1)\\ t_{uw}(\xi,\eta,\zeta) =&(1-\xi)(1-\zeta)\mathbf{x}(0,\eta,0)+(1-\xi)\zeta\mathbf{x}(0,\eta,1)+\xi(1-\zeta)\mathbf{x}(1,\eta,0)+\xi\zeta\mathbf{x}(1,\eta,1)\\ t_{uv}(\xi,\eta,\zeta) =&(1-\xi)(1-\eta)\mathbf{x}(0,0,\zeta)+(1-\xi)\eta\mathbf{x}(0,1,\zeta)+\xi(1-\eta)\mathbf{x}(1,0,\zeta)+\xi\eta\mathbf{x}(1,1,\zeta)\\ t_{vw}(\xi,\eta,\zeta) =&(1-\eta)(1-\zeta)\mathbf{x}(\xi,0,0)+(1-\zeta)\eta\mathbf{x}(\xi,1,0)+\eta(1-\zeta)\mathbf{x}(\xi,0,1)+\eta\zeta\mathbf{x}(\xi,1,1)\\ t_{uvw}(\xi,\eta,\zeta) =&(1-\xi)(1-\eta)(1-\zeta)\mathbf{x}(0,0,0)+(1-\xi)(1-\eta)\zeta\mathbf{x}(0,0,1)\\ &+(1-\xi)\eta(1-\zeta)\mathbf{x}(0,1,0)+(1-\xi)\eta\zeta\mathbf{x}(0,1,1)\\ &+\xi(1-\eta)(1-\zeta)\mathbf{x}(1,0,0)+\xi(1-\eta)\zeta\mathbf{x}(1,0,1)\\ &+\xi\eta(1-\zeta)\mathbf{x}(1,1,0)+\xi\eta\zeta\mathbf{x}(1,1,1)\\ \mathbf{x}(\xi,\eta,\zeta) =&t_{uu}+t_{vv}+t_{ww}-t_{uw}-t_{uv}-t_{vw}+t_{uvw}\end{aligned}$$

Other techniques of grid generation and further information can be found in [12].

3 Coupled thermal FSI with MpCCI, FEAP and FASTEST

3.1 Coupled Fluid-Solid Problems

For many engineering applications, mechanically and/or thermally coupled fluid-solid problems play a decisive role. For thermally coupled problems one has to consider convective heat transfer and fluid properties of the fluid and, on the other side, thermal stresses and mechanical dissipation of the solid. Forces and deformations can be transferred directly between the fluid and the solid.

There are two approaches for coupling a fluid and a structure simulation. As it is possible to solve the equation system of the solid also via the FVM, one could use one single solver for both geometries. Then the coupling would be provided like any other parameter in the equation system. On the other side,one can partition the problem domain in a fluid and a solid problem. Afterwards they have to be coupled by an interface that exchanges the designated parameters. Several commercial software packages provide an "All-in-one" solution where the coupling service and both solvers are included. The user does need to generate specific coupling faces or specific consistent boundary conditions.

But it is also possible to couple two different programs like FASTEST for the fluid-flow and FEAP for the structure analysis, where one program does not *know* about the other one. The interpolation between the different grids on the coupling interface is done via the quasi-standard software MpCCI. The idea is, that both programs solve their problems separately, but with alternating boundary conditions, derived from the results of the other solver, at previously defined coupling faces.

In the following chapters the setup for a coupled fluid-structure simulation with the mentioned software-packages are described. The mathematical background is again based on the fundamental conservation laws and will be described in the next chapter.

3.1.1 Modelling of Coupled Fluid-Solid Problems

The basis is again the fundamental conservation laws for mass, momentum, moment of momentum and energy which are valid for any subvolume V of the problem domain either in fluid or solid parts but with the different material laws and individual needs as described in Chap. 2.

When dealing with the movement of a solid within a fluid, it is convenient to consider the *arbitrary Lagrangian-Eulerian (ALE) formulation*. With this formulation one has the advantages of both formulations. Detailed information see [6].

For the boundary conditions one has to distinguish between different kinds: a solid boundary, a fluid boundary and a fluid-solid interface. The treatment of the solid and fluid boundary conditions are described in the chapters before. At the fluid-solid interface the velocities and the stresses have to fulfil the conditions:

$$v_i = \frac{Du_i^b}{Dt} \quad \text{and} \quad \sigma_{ij} n_j = T_{ij} n_j. \tag{3.1}$$

where u_i^b and Du_i^b/Dt are the displacement and velocity of the interface, respectively. When also heat transfer is involved, the temperatures as well as the heat fluxes have to coincide on the coupling surface.

3.2 Structure

The coupling via MpCCI is based on the exchange of parameters at certain coupling nodes between the two programs. The iterative process can be described by the following steps:

- The fluid solver computes a first time step calculation and sends wall forces and thermal fluxes to MpCCI
- MpCCI interpolates the information to the coupling surface of the solid problem domain
- The structural solver computes temperatures and velocities and sends them back to MpCCI
- MpCCI interpolates the data to the coupling surface of the fluid problem domain
- The fluid solver computes the grid movement

- Check of TFSI convergency. If it is true, the next time step will be computed or another FSI iteration will be done

This is also illustrated in (fig. 3.1) taken from [4].

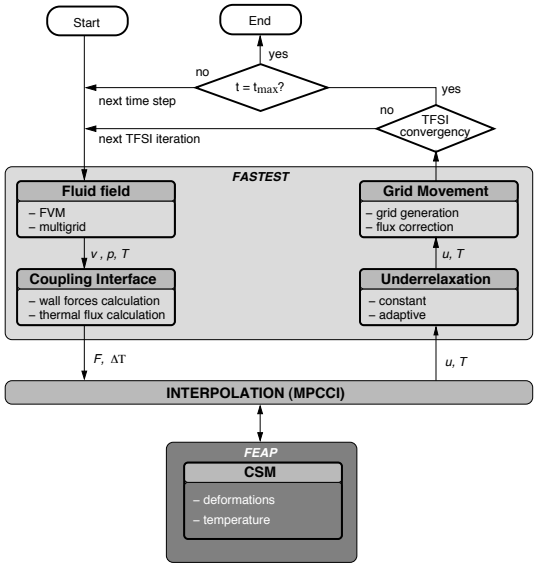


Figure 3.1: Simplified illustration of the work flow of MpCCI

This thesis deals with the definition of the coupling-face in FEAP. As described above, MpCCI needs each node of the coupling surface. For the structural analysis, they can be defined by the node number in the FEAP-input file. For the fluid flow icemCFD can be used as a grid generator to construct the geometry and define the coupling surfaces.

3.3 MpCCI as Coupling Interface

MpCCI provides the coupling of FEAP and FASTEST with just some extra definitions to the usual input file. Both geometries have to be in the same dimensions and the coupling faces have to be the same size. Details about the setup of the FASTEST input file can be found in [2].

3.3.1 FEAP Structure

The FEAP input file consist of two blocks. The first block includes the geometry and material definitions as well as the boundary conditions and initial displacements. For a coupled computation another command for MpCCI has to be placed there. In the old implementation this command included the naming of each coupling node and element. The second block includes the commands for solving the problem and the communication with MpCCI. Further information about the setup of the input file can be found in [9].

3.3.2 Setup of the input files

First we will have a look at the setup of the FEAP input file for a coupled computation. If the problem one wants to simulate has thermal boundary conditions, FEAP simulates them by the use of an extra material that has to be assigned to the boundary domain. This additional material simulates the thermal fluxes at the surface. Its dimension has to be $n-1$ and shall be applied like any other material by the use of the *BLENd* command e.g. and have the following properties:

```
1 MATErial 2      ! or any other free number
2   COUPling
3   PLANe
```

Listing 3.1: Standard coupling material

For an FSI simulation where one is interested in thermal coupling effects, this material has to be assigned on the coupling surface to simulate the thermal fluxes of the fluid.

Then the MpCCI parameters have to be set up and the coupling nodes defined within its own *MPCCI*-block. This can be done by naming each coupling node and element individually:

```
1  MPCCi
2    MESH 20                ! Mesh Parameter
3    PART25                 ! Part ID
4    NODE
5      1    x1 y1 z1
6      2    x2 y2 z2
7      . . .
8    ELEM
9      1 n1 n2 n3 n4 mate
10     2 n1 n2 n3 n4 mate
11     . . .
12 MPENd
```

Listing 3.2: MpCCI definitions

or, as later described, automatically with the new functions. Hence the *NODE* command gets redundant and a new command has to be placed in the the *MACRo* at the end.

The last step is to modify the *MACRo* command. First of all the MpCCI has to be initialised (line 4) what makes FEAP to send the information defined in the *MPCCi* command (3.2) for the coupling nodes and coupling elements to MpCCI. This is followed by a *LOOP* command with a desired amount of timesteps. In each timestep another *LOOP* command is necessary for the fsi iterations. As an example the *MACRo* in (listing 3.3) computes one timestep (line 6) and 50 fsi iterations (line 8). This loop contains the sending and receiving of the desired parameters and the solving of the solid problem domain (line 11-13) by a linear newton solver.

```
1  MACR
2    NOPRint
3    DT,,1.0e10
4    MPCCI,INIT             ! MPCCI initialised and parameters are sent
5    TRANsient,BACK
6    LOOP,ts,1              ! 1 timestep
7      TIME
8      LOOP,fsiiter,50      ! 50 fsi iterations
9        RECV,HEAT          ! temperatures from FASTEST are received
10       RECV,FORC          ! forces from FASTEST are received
11       LOOP,newton,300    ! solving
12         TANG,,1
13       next,newton
14       SEND,TEMP          ! newly calculated temperatures are send to FASTEST
15       SEND,DISP          ! newly calculated displacements are send to FASTEST
16       CONVergency        ! Check for convergency
17     next,fsiiter
18   next,ts
19   MPCCI,end              ! Close the MPCCI Server
20 END
```

Listing 3.3: a Macro for FSI

This data needs to be saved and named with a trailing *i* and no file extensions, e.g. *iprojectname*. Another file has to be written, where the paths to the log and output files are declared. It should be named just with the project name and could look like the following:

```
1
/home/user/FEAP75/projects/iprojectname
/home/user/FEAP75/projects/Oprojectname
/home/user/FEAP75/projects/Rprojectname
/home/user/FEAP75/projects/Pprojectname
y
```

Listing 3.4: Pathdefinitions for FEAP

This is all that needs to be done for preparing a FEAP input file for a coupled FSI-computation for the FEAP-part. The changes needed in the input files of FASTEST are note described here but can be found in [2].

4 Implemented functions

As one can see in 3.2, MpCCI needs the number of each coupling node in order to define the coupling surface. As they can change by refining the mesh or editing the geometry, the node definitions have to be updated as well. Quite often a coupling surface has several thousand coupling nodes and the need for an automatisation of this procedure gets obvious. This is done in this work by the implementation of several functions for different geometries and will be explained in details in the following chapter. Afterwards the functions are validated by examples.

4.1 Installing the functions

The following instruction describes how all functions that have been implemented are installed. We assume, that one already has FEAP properly installed (information about the installation of FEAP can be found in [8]) and has a basic knowledge about user functions and their implementation. Further details about how to adopt *usermacros* can be found in [10]. The extensions have been tested in FEAP Ver. 7.5.

The extension comprises several newly developed files:

File	Purpose
umacr6.F	Usermacro that provides the FEAP-commands for calling the functions for planar, circular or conical surfaces. Of course it can be numbered differently if 6 is already in use.
umacr7.F	Usermacro that provides the FEAP-commands for calling the functions for freeform coupling surfaces. Can of course also named with any other free number.
boundary8brickelem.F	This file provides the finding of the orientation as well as organising the map for the coupling nodes which is described later.
definesurfcbc.F	Includes the routines for finding coupling nodes on planar surfaces.
definepolcbc.F	Includes the routines for finding coupling nodes on circular coupling surfaces, e.g. a tubeflow
definediffcbc.F	Includes the routines for finding coupling nodes on conical coupling surfaces like diffusors
definetfisurface.F	Includes the routines for finding coupling nodes on freeform coupling surfaces
TFI.F	Transformation from logical coordinates to real coordinates by the use of TFI
functions.F	Subroutines for the TFI coupling
defineSpline.F	Subroutine that conditions the user-definitions of a spline from the FEAP input file
alloctfiarray.F	Subroutine that allocates the needed arrays for the TFI

Table 4.1: List of all needed files

All these files have to be put in the

`/FEAP75/src/user/MPCCI/`

folder. Afterwards FEAP has to be compiled and the following functions can be used. The sources can be found in the Appendix C-M.

4.2 General Workflow

The implemented functions have a common general workflow, which will be described here. The user defines a coupling surface by naming parameters in the FEAP input file that describe the coupling surface uniquely. These can be points or radii, or support points for the TFI. This data will be edited as arrays to be used in the routines. Then each node of the geometry will be considered elementwise and checked whether it is a coupling node or not by computing its distance to the coupling face. If it is less than δ (parameter dif in the sources), it gets identified as a coupling node. As only eight nodes brick elements are considered here, a coupling element always has four coupling nodes. Since the nodes are considered element wise, the founded node number is the local one and needs to be transferred to global. This is done by the use of the orientation of the element. Then together with the element number one can get the global node numbers out of the nodes array. After each element has been checked, the coupling nodes and elements are stored in an array and sent to MpCCI.

That means that still each node and element are itemized, but the recognition is automated.

4.3 Implemented Functions

4.3.1 Planar Coupling Surfaces

A typical problem geometry in research and industry is a plane coupling surface, for example a fluid acting on a straight wall. The function *PLAN* provides the recognition of coupling nodes on a plane surface just by the use of three points ($P_1(x_1, y_1, z_1), P_2(x_2, y_2, z_2)$ and $P_3(x_3, y_3, z_3)$) of the surface. FEAP then transforms the given points into the *Hessian normal form* and computes the distance between the defined plane and every node of the grid:

$$dis = \frac{|\,(v_1 - a_1) * n_1 + (v_2 - a_2) * n_2 + (v_3 - a_3) * n_3\,|}{\sqrt{n_1^2 + n_2^2 + n_3^2}} \tag{4.1}$$

where $\vec{V}(v_1, v_2, v_3)$ is the vertex which distance shall be computed, $\vec{A}(a_1, a_2, a_3)$ the position vector of one point of the surface and $\vec{N}(n_1, n_2, n_3)$ the normal vector of the plane. If dis is less than δ, the node gets buffered as a coupling node.

The mentioned parameters (P_1, P_2 and P_3) have to be made available in the FEAP input file, after the last *END* command:

```
PLAN x1 y1 z1 x2 y2 z2 x3 y3 z3
```

This will be read after the command *FCBC,init* in the *MACRo* section is processed by FEAP.

4.3.2 Cylindrical Coupling Surfaces

In research topics, very often one has to consider tubeflow analysis. They are also theoretically well known and can be compared to computations made manually. The functions can of course also be used for a plate e.g., where the fluid moves around as in 4.4.6. The function *POLA* has been developed for an easy recognition of coupling nodes on a cylindrical surface.

The workflow of finding coupling nodes in this case is based on computing each node's distance of the geometry to the coupling surface.

Again a node is identified as coupling node, when the computed distance is less than δ. In order to make FEAP to search for the coupling nodes, the command *FCBC,init* has to be placed in the very beginning of the *MACRo*. Then FEAP reads the lines after the last *END* command where one has to provide the needed definitions for the coupling surface

- Two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$, defining the axis
- The direction d (1, 2 or 3) of the axis,
- The radius r.

by the use of the command

```
POLA x1 y1 z1 x2 y2 z2 r d
```

The determination of the distance is done by the computation of the distance of a point $\vec{T}(t_1, t_2, t_3)$ to a straight line (axis) defined by the two points (P_1 and P_2):

- Computation of the direction vector $\vec{v}$ given by P_1 and P_2.
- Construction of a plane through the testingpoint $\vec{T}$ whereas the normal vector is $\vec{v}$.
- Computation of the intersection P' between the plane and the axis.
- Computation of the distance between the two nodes P' and T.

4.3.3 Coned Coupling Surfaces

Very often when dealing with tubeflow problems, the profile gets reduced or expanded. These geometries are known as diffusers and are often considered in research and industry too. The function *DIFF* provides the automatic recognition of coupling nodes on coned surfaces like diffusers.

The selection of the coupling nodes here is similar to the selection at circular faces with the difference, that the distance of the coupling nodes to the axis differs at each point of the axis. To recognise the correct nodes, the expected distance of a coupling node is compared to the actual. As we know the x_1 position of the testing node $T(x_1, y_1, z_1)$, and the points R_1 and R_2 we also know its x_2 position and therefore the expected distance. This shall be illustrated in fig. 4.1.

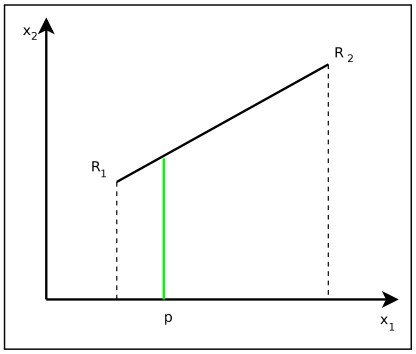


Figure 4.1: Computation of the expected distance for coned surfaces

The needed parameters here are

- Two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$ defining the axis,
- The origin or (1, 2 or 3) of the axis,
- The two radii (r_1 and r_2),

and have to be placed after the *MACRo* by the use of the command

```
DIFF x1 y1 z1 x2 y2 z2 or r1 r2
```

4.3.4 Freeform Coupling Surfaces

Sometimes it is not possible to define a surface just by the use of the geometries described above. The TFI can be used in order to describe freeform surfaces. In this case, the user only has to provide functions for the curves surrounding the surface. The implemented functions provide a definition of the curves by the use of a linear interpolation or cubic splines, which could also be easily expanded.

Here the identification of a coupling node is a little bit different: first a logical mesh, with the definitions of the input file is generated. This mesh is transferred via TFI into a physical mesh, which should match the coupling surface of the geometry. The coordinates of these nodes are stored. Then FEAP computes the difference (distance) of a testing node to each node of the TFI surface. If it is less than δ it is identified as a coupling node.

In details: first of all the input data of the spline definitions get transferred in order to get an equation with which each point of the spline can be computed. As also non equidistant grid points are allowed, first the length of i-th interval is determined. Then a linear equation system with the equations of each interval is set up. As we have cubic splines here, we have four unknowns. For each spline we have two points (the starting and ending point) and the derivatives as well. Therefore each spline can be identified exactly.

The next step is the TFI where the logical mesh is transferred to the physical mesh. This is done by the use of the equation described in 2.8. As a result we have the physical coordinates that are buffered.

FEAP checks every node's distance to the surface, buffers matching nodes and stores their global coordinates for sending to MpCCI.

The parameters the user has to provide are

- The coordinates of the vertices $\vec{v_1}$ - $\vec{v_8}$,

- The curves C_1 - C_8,
- The type of each curve (linear or spline).

To make FEAP to read the TFI definitions, instead of *FCBC,init* one has to write *TFIC,init* in the very beginning of the *MACRo*. The parameters are provided in following configuration (again placed after the *END*:

```
NODE,8              ! 8 nodes
 1 x1 y1 z1         ! nodenumber and coordinates
 4 x4 y4 z4
...
```

Before defining the curves, the user has to specify the discretisation and total amount of curves as well as the surface that will be described via TFI.

```
CURV,ncurve,dis1,dis2,nTFI   ! number of curves, discretisation in 1-direction,
                             ! discretisation in 2, number of TFI-Surface
```

The numbering (nTFI) can be found in (fig. 4.2):

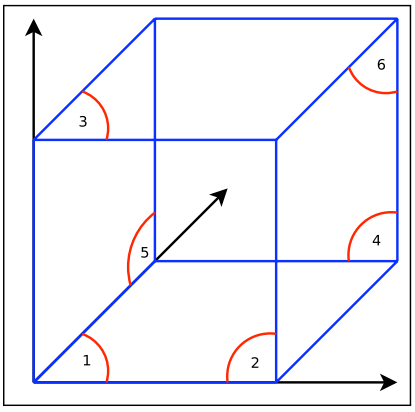


Figure 4.2: Numbering of the box - Needed to define the TFI Surface

When defining a curve, the user has to provide the node numbers (nn) of the vertexes (n_1 and n_2) that are connected and whether it is linear (1) or a spline (3). If a spline should be defined, the user has to send more parameters. These are the number of support points (n_sp) of the spline, the *initial* and *final bending* (in_b and fi_b) and the axis of the constant values (ax_1) and the axis with the variable values (ax_2):

```
nn n1 n2 1                          ! Linear connection of node 1 with node 2
nn n1 n2 3 n_sp in_b fin_b ax1 ax2  ! Spline between node n1 and n2
```

The coordinates (x_s, y_s, z_s) of the support points have to be placed directly after a spline definition command. The numbering (spn) of the following support points has to start with 1 again:

```
spn xs ys zs
```

A whole numerical example will be presented in (4.4.4).

4.4 Examples

4.4.1 Planar example

As an example we will have a look at a bevel cube illustrated in figure 4.3. The solid is on the left side and bevelled, the fluid on the right hand side.

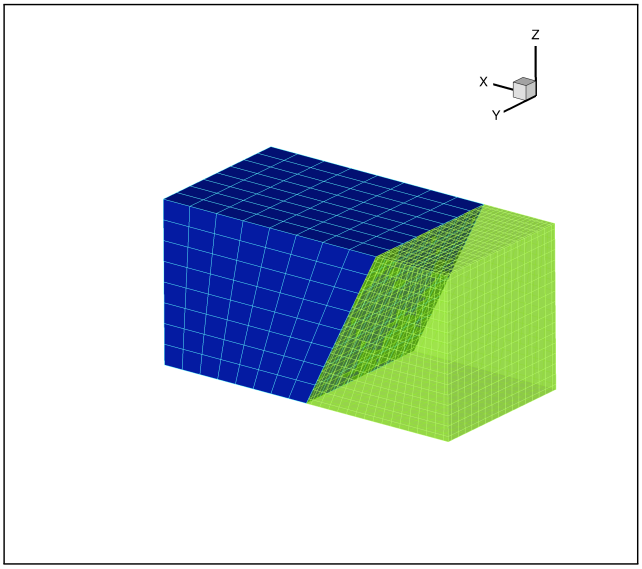


Figure 4.3: Example of a planar coupling surface

The input file for this geometry is:

```
FEAP
0 0 0 3 4 8

PARAmeter
  nx=8
  ny=8
  nz=8
  x1=0.1
  y1=0.0
  z1=0.0
  x2=0.2
  y2=0.1
  z2=0.1
  x3=0.05

BLOCK
  CARTesian nx ny nz 0 0 1 10
    1 x1 y1 z1
    2 x2 y1 z1
    3 x2 y2 z1
    4 x1 y2 z1
    5 x3 y1 z2
    6 x2 y1 z2
    7 x2 y2 z2
    8 x3 y2 z2

BLOCK
  CARTesian ny nz 0 0 2 1 0
    1 x1 y1 z1
    2 x1 y2 z1
```

```
31      3 x3 y2 z2
32      4 x3 y1 z2
33
34  END
35
36  TIE
37
38  MACR
39    FCBC,INIT                ! read the Command after END
40    ...
41  END
42
43  PLAN x1 y1 z1 x1 y2 z1 x3 y2 z2
```

Listing 4.1: Input file for planar coupling faces

First the parameters (measures and discretisation) are set. Afterwards the *BLOCK* command with the eight nodes creates the box and another *BLOCk* command sets the material 2 to the coupling surface for using a thermal flux boundary condition. Other definitions like the materials, displacements, boundaries are taken out for clearance. In the last step the *MACRo* calls the FCBC command and the parameters after *END* are read, where three of the four vertices of the coupling plane are set.

The velocities u, v, and w of the fluid were set to zero, so just a stationary heat transfer between the solid and the fluid is considered here. The solid is modelled as a thermo mechanical isotropic body with the thermal diffusivity $\lambda = 45W/m^2K$ (steel) and the fluid with a heat conductivity of $\lambda = 0.600176W/m^2K$ (water). The heat from the left side of the solid gets transferred to the coupling surface and there it heats up the fluid. So one can see the convective heat transfer which qualitatively looks normal due to the greater heat conductivity in the solid (fig. 4.4).

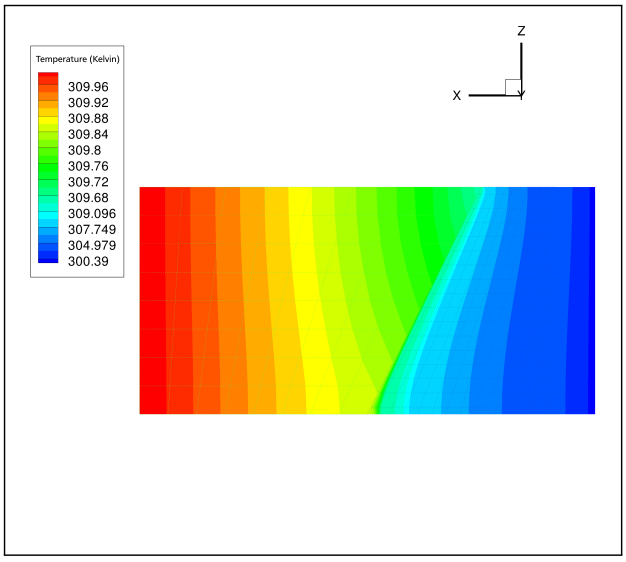


Figure 4.4: Results of the planar coupling: Temperature plot of the geometry in Kelvin - Slice in the x-z-plane through the middle of the geometry

4.4.2 Cylindrical example

As an example of a cylindrical coupling surface we will have a look at a tube with a fluid inside (see fig. 4.5). Again just a stationary heat transfer is considered here. The tube is 2 metres long, the inner radius is 1 metre, the outer 2 metres:

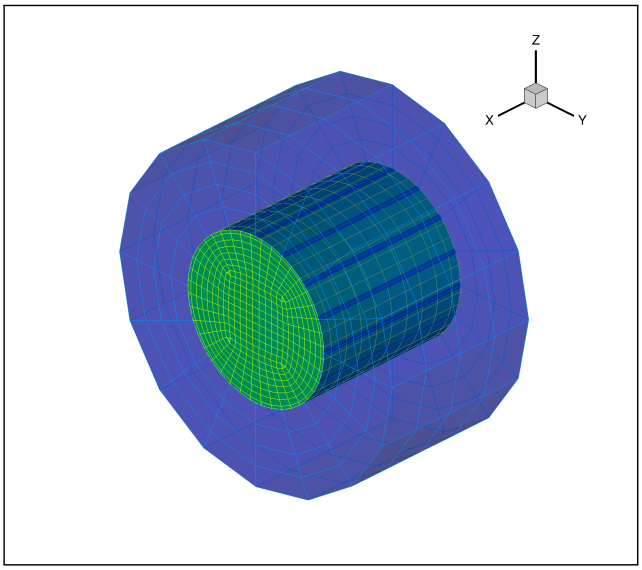


Figure 4.5: Example of a polar geometry

For the computation, the outer surface of solid has an initial temperature of 200°K, the Fluid 300°K.
The input file of the FEAP geometry:

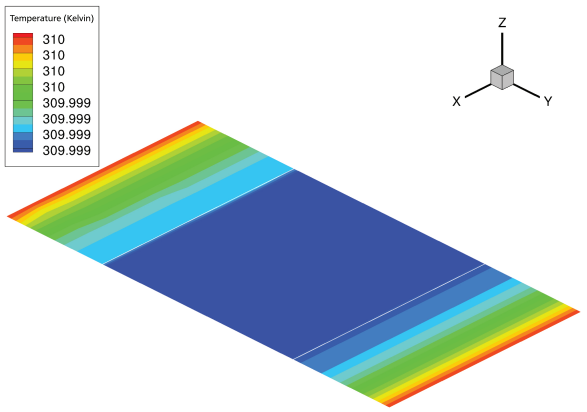
```
FEAP
0 0 0 3 4 8

SNODes
   1 0 0 0
   2 0 0 1
   3 0 1 0
   4 0 0 2
   5 0 2 0
   6 2 0 1
   7 2 1 0
   8 2 0 2
   9 2 2 0
   10 2 0 0
   11 0 0 -1
   12 0 0 -2
   13 2 0 -1
   14 2 0 -2
   15 0 -1 0
   16 0 -2 0
   17 2 -1 0
   18 2 -2 0

SIDE
   POLAr 3 2 1
   POLAr 5 4 1
   POLAr 7 6 10
   POLAr 9 8 10
   POLAr 3 11 1
   POLAr 5 12 1
```

```
31    POLAr 7 13 10
32    POLAr 9 14 10
33    POLAr 11 15 1
34    POLAr 12 16 1
35    POLAr 13 17 10
36    POLAr 14 18 10
37    POLAr 15 2 1
38    POLAR 16 4 1
39    POLAr 17 6 10
40    POLAr 18 8 10
41
42  BLENd 1
43    SOLId 8 8 8 0 0 1
44      2 4 5 3 6 8 9 7
45
46  BLENd 2
47    SOLId 8 8 8 0 0 1
48      12 11 3 5 14 13 7 9
49
50  BLENd 3
51    SOLId 8 8 8 0 0 1
52        16 15 11 12 18 17 13 14
53
54  BLENd 4
55    SOLId 8 8 8 0 0 1
56        16 4 2 15 18 8 6 17
57
58  BLENd 5
59    SURFace 8 8 0 0 2
60      2 3 7 6
61
62  BLENd 6
63    SURFace 8 8 0 0 2
64        3 11 13 7
65
66  BLENd 7
67    SURFace 8 8 0 0 2
68        11 15 17 13
69
70  BLENd 8
71    SURFace 8 8 0 0 2
72        15 2 6 17
73
74  END
75
76  TIE
77
78  MACR
79    FCBC,INIT
80    ...
81  END
82
83  POLA,0,0,0,2,0,0,1,1
```

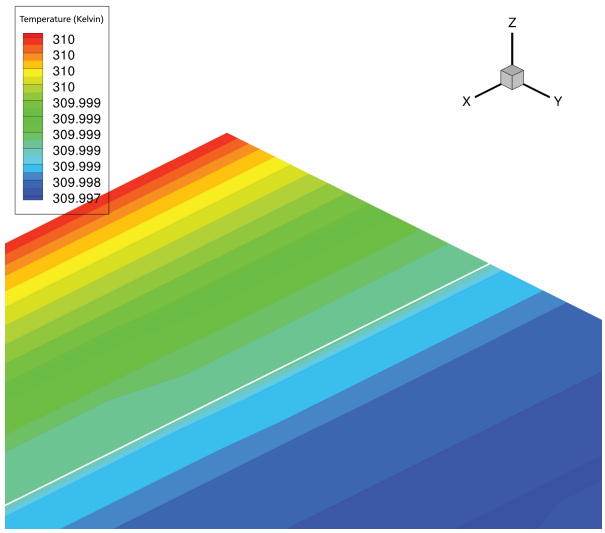
Listing 4.2: Input file for cylindrical coupling faces

First we have a definition of *supernodes* that are used for the polar curve definitions afterwards. A *POLAr n1 n2 c* command creates a circular side between $n1$ and $n2$ with the centre c. As this works only for a quarter of a circle 16 commands (eight for each side) are necessary. Then the *BLENd - SOLId* command creates solid body by the use of 8 nodes

and *BLEND - SURFace* creates surfaces with Material 2 to simulate the heat conductivity there. In the *MACRo* again the *FCBC* command makes FEAP to read the last lines where the parameters for the coupling surface are found.



(a) Overview - Slice in the x-y-plane through the middle of the geometry



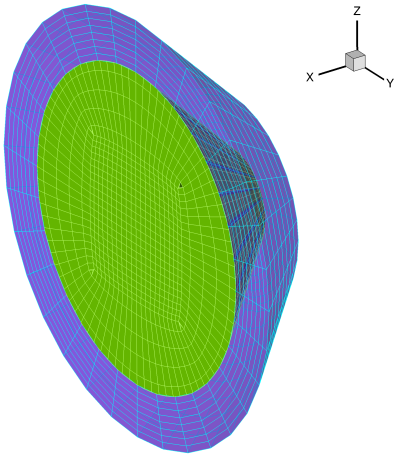
(b) Detail inlet and diffuser - Slice through the x-y-plane in the middle of the geometry

Figure 4.6: Results of the tubeflow - Temperature plot in Kelvin

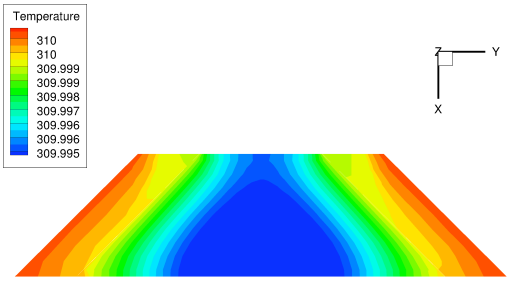
The material and fluid properties were the same as in the planar example (steel and water). One can see, that the solid near the coupling face heats up and the fluid cools down respectively and the coupling succeeded. The temperature profile qualitatively looks normal as the heat conductivity in the solid is greater.

4.4.3 Coned example

As an example we will have a look, at a simple coned geometry with a cold fluid inside and a hot solid around it illustrated in fig 4.7. The smaller radius is 1 metre, the greater 3 metres and its total length is 2 metres. The fluid's initial temperature is 300 °K and the outer surface of the solid is 310°K and again the same material/fluid properties are used. We consider stationary heat transfer here. One can see in (b) that the fluid heats up towards the solid whereas the solid cools down and the coupling succeeded. The fluid near the coupling surfaces heats up on quite a small distance whereas the heat transfer through the solid is widespread over a longer distance due to the greater heat conductivity.



(a) Overview of the geometry: fluid surrounded by solid



(b) Temperature plot of a Slice in the x-y-plane in the middle of the geometry in Kelvin

Figure 4.7: Geometry and results of the coned example

The FEAP input file is nearly the same as in 4.4.2 just with different coordinates of the supernodes and is therefore not presented completely. The greater radius of the Diffuser is 3 metres, the other 1 metre as before. Therefore the parameters for the coupling functions are:

```
1  FEAP
2  0 0 0 3 4 8
3
4  SNODes
5     6 2 0 3
6     7 2 3 0
7     8 2 0 4
8     9 2 4 0
9     13 2 0 -3
10    14 2 0 -4
11    17 2 -3 0
12    18 2 -4 0
13    ...
14
15 END
16
17 MACR
18    FCBC,INIT
19    ...
20 END
21
22 DIFF,0,0,0,2.1,0,0,1,1,3
```

Listing 4.3: Input file for freeform coupling faces

4.4.4 Freeform example

In order to use a freeform geometry the user has to provide more data to specify the the geometry. One has to define all nodes and curves together with needed support points for the splines. This shell be illustrated by the use of the figure fig. 4.8.

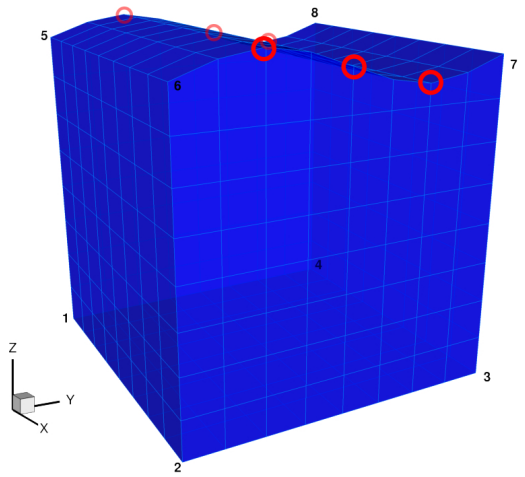


Figure 4.8: Example of a freeform geometry

The geometry is a box with a rolling top. The edges are of a length of four metres. The spline definitions were provided by three support points, equally spaced. The highest point is 4.25 metres high, the lowest 3.75 metres marked with a circle in fig. 4.8. The nodes are numbered as shown, the TFI surface number is 3.

In order to describe this geometry, the input file has to look as shown in 4.4.

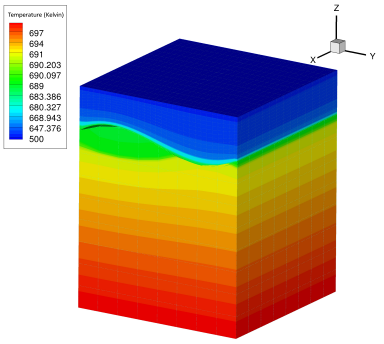
```
1  FEAP
2  0 0 0 3 4 8
3
4  COORdinates
5    1 1 0.0 0.0 0.0
6    9 0 4.0 0.0 0.0
7    10 1 0.0 0.5 0.0
8    18 0 4.0 0.5 0.0
9    ...
10
11 ELEMents
12   1 0 1 1 2 11 10 82 83 92 91
13   2 0 1 2 3 12 11 83 84 93 92
14   ...
15   513 0 2 649 650 659 658
16   514 0 2 650 651 660 659
17   ...
18
19 END
20
21 TIE
22
23 MACR
24   TFIC,INIT
25   ...
26 END
27
28 NODE,8
```

```
1 0 0 0
2 4 0 0
3 4 4 0
4 0 4 0
5 0 0 4
6 4 0 4
7 4 4 4
8 0 4 4
CURV,12,9,9,3
1 1 2 1
2 5 6 1
3 8 7 1
4 4 3 1
5 1 4 1
6 2 3 1
7 6 7 3 3 0 0 2 3
4 1 4.25
4 2 4
4 3 3.75
8 5 8 3 3 0 0 2 3
0 1 4.25
0 2 4
0 3 3.75
9 1 5 1
10 2 6 1
11 3 7 1
12 4 8 1
```

Listing 4.4: Input file for freeform coupling faces

The geometry is defined nodewise and each element individually. Therefore the input file has been shortened. What we can see in the definition of the coupling parameters are the splines connecting node 6 with 7 and 5 with 8, respectively, together with the support points (marked with circles in fig. 4.8) afterwards.

If we now assume a cold fluid (500°K) (water) on the top of the surface (see Fig. 4.8), and a temperature of 700°K at the bottom of the solid (steel) we expect a similar behaviour like the other geometries. The geometry of both, the fluid and the solid are presented in fig. 4.8 and the results in fig. 4.9



(a) Result of the coupled computation - Temperature plot in Kelvin

Figure 4.9: Results of the tube

As one can see, the heat is transported from the bottom of the solid to its top, and then transferred to the fluid above. The coupling succeeded which can be qualitatively seen in fig. 4.9

4.4.5 Comparative Simulation

In order to evaluate the functions, a former computation which was made with the node- and elementwise, definition should be compared to a computation using the developed functions. The geometry, boundary and material definitions were identical, just like the fluid properties. Only the MpCCI definitions in the FEAP-input file were edited. What we expect is the same amount of coupling nodes and elements and therefore identical solutions.

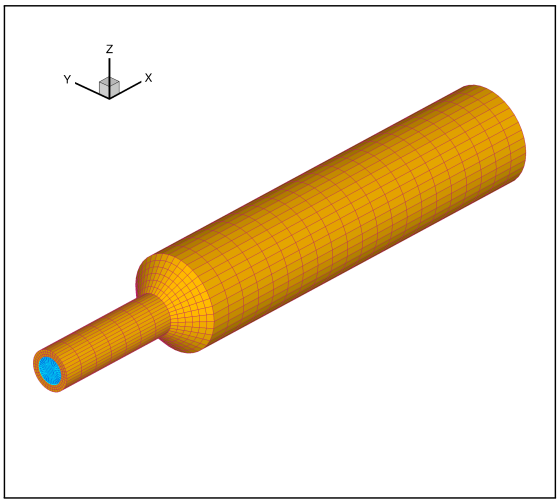


Figure 4.10: Geometry - Solid diffuser with fluid inside

The geometry in figure 4.10 is a tubeflow with a diffuser. The outer geometry is the solid, the inner the fluid. The origin is in the centre of the the small radius on the outer side. It has a length of 0.1 metre and its radius is 0.01 metre. The diffuser has a length of 0.02 metre and ends with a radius of 0.03 metre. The larger tube has a length of 0.3 metre.

The coupling surface is the whole inner surface of the solid and the outer surface of the fluid respectively. We have an inlet flow at the side with the smaller radius and a outlet at the other side. The velocity at the inlet in x-direction is 0.073 m/s, no velocity in y- and z-direction. The temperature is 1000°K at the inlet boundary. The solid tube has a initial temperature of 450 K. The flow has been modelled with the use of the k-ϵ-model. The Reynoldsnumber was 5000. All other fluid properties can be found in appendix [A].

Now we will have a look at the FEAP input files. The geometry definitions derive from an export of a CAD Tool. Each node and element is defined individually. The coupling material was defined element wise and the boundary conditions also node wise. As these are several thousand lines of code, they are not presented here.

The definitions for the coupling surface was in the former input file a list of all coupling nodes. Now the functions *POLA* and *DIFF* can be used to recognise the coupling nodes automatically:

```
1  MACR
2    NOPRint
3    DT,,1.0e10
4    FCBC,INIT
5    MPCCI,INIT
6    TRANsient,BACK
7    LOOP,ts,1
8      TECP,init,1
9      TIME
10     LOOP,fsiiter,50
11       RECV,HEAT
12       RECV,FORC
13       LOOP,newton,300
14         TANG,,1
15       next,newton
16       SEND,TEMP
17       SEND,DISP
18       CONVergency
19     next,fsiiter
20     TECP,write,1,5600,4080
```

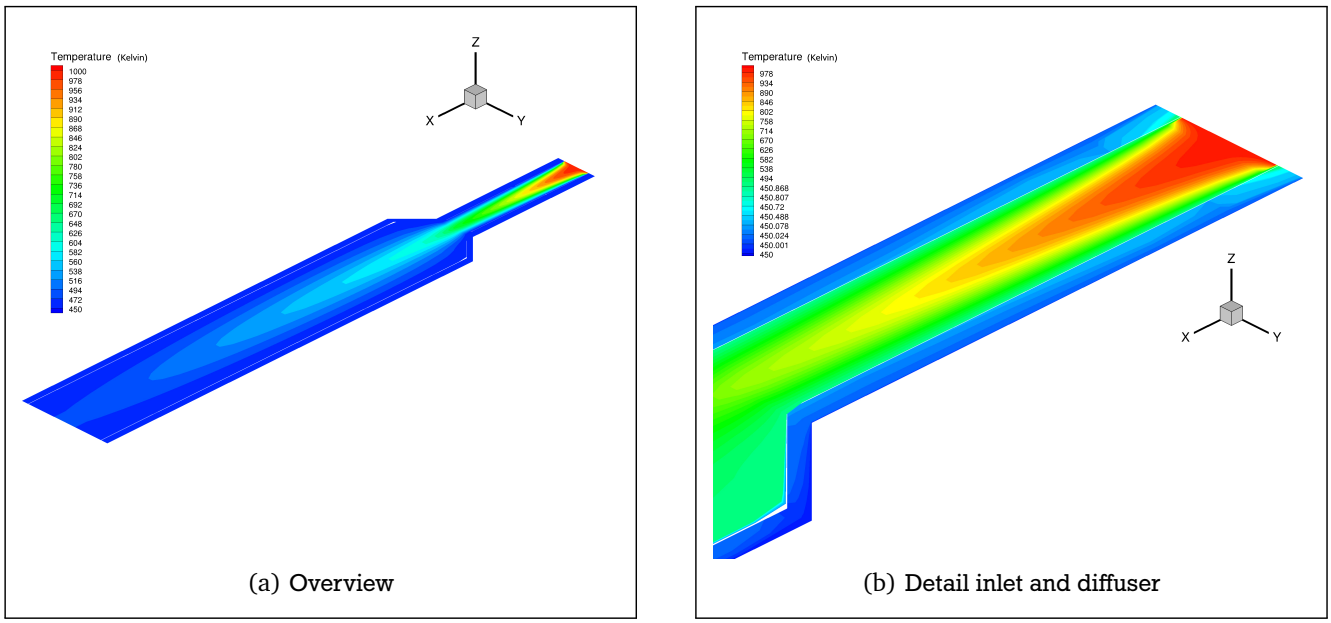
```
21        TECP,close,1
22      next,ts
23      MPCCI,END
24  END
25
26  POLA,0,0,0,0.1,0,0,0.01,1
27  DIFF,0.1,0,0,0.12,0,0,1,0.01,0.03
28  POLA,0.12,0,0,0.42,0,0,0.03,1
```

Listing 4.5: MACRo for the tubeflow with diffuser

As it is a known quirk of FEAP, one should leave some clear lines after the last definition to make it work. That is all that has to be changed in the input file.

As the computation will not start, as long as MpCCI has not a valid partner for each node, one can say, that as soon as MpCCI does not report any errors, the coupling succeeded. The amount of founded coupling nodes and coupling elements can be found in the *feap.out*-file and can be compared to the amount of nodes in the former declaration or at least check if it is in a realistic range. The *FCBC*-functions founded 1400 coupling nodes and 1360 coupling elements, which is exactly the same amount as in the former definition. Of course it may be that the computation diverges in case of inconsistent definitions, but this is not matter of this thesis.

To illustrate the coupling we will have a look at the results of the computation in fig. 4.11. They are identical to the solutions with the former coupling surface definition. Therefore one can say, that the functions provide the expected functionality.



(a) Overview

(b) Detail inlet and diffuser

Figure 4.11: Results of the tubeflow with diffuser - Temperature plot in Kelvin

As one can see in image (b), the solid heats up at the inlet where the inlet flow is hot (1000°K). Another typical behaviour can be found in the transition from the diffuser to the greater tube. The solid does not heat up as the hot flow does not reach this part. Additionally one can see, that the the fluid cools down at the coupling surface.

4.4.6 Incident Flow on both Sides of a Plate

Another example where the functions are used is a computation of my supervisor P. Pironkov. He analysis the incident flow on both side of a plate, whereas the fluid is hot on one side and cold on the other. Image 4.12 should illustrate the geometry. The plate in the centre is the solid geometry provided by FEAP. The box simulates a infinite stretched room as all sides have adiabatic boundary conditions. The inlet flows were applied in the two tubes, positioned at the same level as the plate.

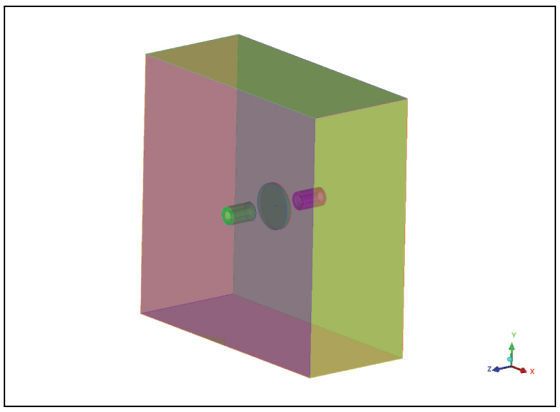


Figure 4.12: Geometry - Incident Flow an a Plate

The velocity at the inflow is 26.1 m/s and -7.19 m/s, the temperature 640°K and 294°K respectively. As turbulence model the k-ϵ-model has been adopted as well with a Reynoldsnumber of the fluid of 5000.

The used FEAP input file can be found in appendix [B]. In order to avoid bad elements a block structured mesh has been applied here. That was implemented by the use of five blocks. One cuboidal block in the centre and four blocks around it whereas the outer sides are circular and the inner straight. The *POLA* command provides the recognition of the coupling nodes on the cylindrical surface whereas the *PLAN* commands recognise the coupling nodes on the plane surfaces where the flow impacts.

As the computation is still in progress no results are presented here but the coupling succeeded as the computation started successfully.

5 Conclusion

The task of the present Bachelor Theses was to implement various functions that provide an automatic recognition of boundary conditions. The developed functions should be applied for the use of FEAP and FASTEST in thermal FSI by the use of the coupling interface MPCCI.

This was done for simple geometries like plane, polar or coned as well as free-form surfaces by the use of some user provided parameters.

5.1 Evaluation

As shown in 4 the functions provide an automatic recognition of coupling nodes for the mentioned geometries. In Chapter 4.4, the multiple usage of the commands have been proved. Therefore one can say, that the implemented functions fulfil the requirements. A great advantage now is, that in case of a mesh refinement or a change in the geometry, the definitions of the coupling surfaces do not have to change. The user is more flexible in the use of the solid geometry. As MpCCI will not start a computation if not all coupling nodes are well defined, one can say that coupling succeeded as soon as the computation starts. The results then do not depend on the used functions, but on the boundary conditions of the solid and the properties of the fluid flow.

By the use of freeform surfaces one can also describe several more complicated surfaces that can be described by the TFI. If a geometry does not fit any provided functions, new routines can be implemented straight forward or existing routines can be extended.

5.2 Prospect

The implemented functions can easily be extended if for example a spline definition for the TFI does no match the given geometry or cannot be described by any function. Also other geometries like spherical elliptical ones could be implemented as standard geometries.

As the boundary and displacement definitions still need to be set nodewise, the implemented functions could be used for these definitions as well. The implementation would be straight forward and be very useful for the user. Then FEAP would again be a powerful tool for FSI.

Another feature of the implemented functions are, that the functions developed for the TFI can also be used for grid generation. The geometry in 4.4.4 has been created by the use of the implemented functions. Hence with a few modifications a TFI grid generator could be adopted as well.

Bibliography

[1] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1993.

[2] Fachgebiet fuer numerische Berechnungsverfahren - TU Darmstadt. *FASTEST Manual*.

[3] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer Verlag, 2001.

[4] P. Pironkov. Dissertation in progress, 2009.

[5] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[6] M. Schäfer. *Computational Engineering Introduction to Numerical Methods*. Springer Verlag, Berlin, 2006.

[7] D. C. Sternel and S. Meynen. *Die Methode der finiten Elemente in der Strukturmechanik*. 2008.

[8] R. L. Taylor. *FEAP – A Finite Element Analysis Program - Version 7.5 Installation Manual*. University of California at Berkeley - Department of Civil and Environmental Engineering, Berkeley, November 2003.

[9] R. L. Taylor. *FEAP – A Finite Element Analysis Program - Version 7.4 User Manual*. University of California at Berkeley - Department of Civil and Environmental Engineering, Berkeley, October 2004.

[10] R. L. Taylor. *FEAP – A Finite Element Analysis Program - Version 7.5 Programmer Manual*. University of California at Berkeley - Department of Civil and Environmental Engineering, Berkeley, September 2004.

[11] P. Wriggers. *Nichtlineare Finite-Element-Methoden*. Springer Verlag, 2001.

[12] S. Yigit. *Phaenomene der Fluid-Struktur-Wechselwirkung und deren numerische Berechnung*. Shaker Verlag, 2007.

A FASTEST id-file

The FASTEST id-file of the coupled FSI simulation tubeflow with diffuser. All parameters are well commented.

```
#$ver02.01
### title
Name des Projektes o.ae.
### read restart
0
### write output
0 0 1          ;restart ascii visual  (0 = no output)
### surface grid data output
                        ;[bc][con]
### file formats for visual output
tecplot6                ;[tecplot6|tecplot6temp]
                        ;[plot3d|plot3dtemp]
### write additional output
0 0 0 0 0 0             ;louts lsurf ldebug lboundc llagr lvogd
### visual output variables
vel pres  temp          ;[vel][pres][tke][edis][temp][conc]
                        ;[den][vis][vism][hcap][hcon][hconm][diff][td]
                        ;[vort][avg][tau][yplus][dist]
### ascii output variables
                        ;[velx][vely][velz][pres][tke][edis][temp][conc]
                        ;[den][vis][vism][hcap][hcon][hconm][diff][td]
                        ;[flux1][flux2][flux3]
### ascii output geometry
                        ;[x][y][z][xc][yc][zc][vol]
                        ;[fx][fy][fz][ar(1-3)(x-z)]
### ascii output of surface data
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
### lcalc
vel temp time          ;[vel][turb|visles][temp][time]
                       ;turb:   use a two-equation turbulence model
                       ;visles: perform a Large Eddy Simulation
### turbulence model
keps74                  ;according to the above choice of [turb|rsm|visles]
                        ;[keps74|kl93|rng92|chien82] for k-eps type models
                        ;[smag|germ]                 for Large Eddy Simulations
### scalar flux model
ls                ; [hp|ls|jm]
### parameters for large eddy simulation
2                 ;[1|2]  determine filterwidth by cell [vol|area]
7                 ;[7|27] number of adjacent cells used for test filtering
0.5               ;underrelaxation for germano parameter
### subgrid scale variance model
equilib       ;[equilib|scalesim|res_var]
### grid levels
1 1                     ;no. of coarse / fine grid
### gravity
0.  0.  0.
### geometric scale
1.d-3 1.d-3 1.d-3
### check for negativ volumes
t
```

```
### small
1.e-30
### residuum limits
1.e-4  1.e+10
### interpolation method
cds                          ;[cds|tbi]
### flux blending
0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.
;vel k eps temp conc mix restr mixvar srs scfl ildm ildmv rf cst vofr
1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0. ;2. Grid
1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0. ;3. Grid

### underrelaxation
0.6 0.6 0.6 0.5 0.9 0.9 0.5 0.3 0.2 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.3 0.9 0.9
;u v w p k eps vis den t c mix restr mixvar srs scfl ildm ildmv rf cst vofr
0.5 0.5 0.5 0.5 0.9 0.9 0.5 0.3 0.1 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.3 0.9 0.9
; 2. Grid
0.3 0.3 0.3 0.5 0.9 0.9 0.5 0.3 0.1 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.3 0.9 0.9
; 3. Grid
### sipsol
0.92                                    ;alfa
0.5 0.5 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
;sor: u v w p k eps t c mix restr mixvar srs scfl ildm ildmv rf cst vofr
1 1 1 5 1 1 5 1 1 1 1 1 1 1 1 1 1            ;nsw: u v w p k eps t
c mix restr mixvar srs scfl ildm ildmv rf cst vofr
### multigrid cycle definition
10000
7 20 1
10 15 30 10 5
### number of multigrid cycles
100 100 100
### time discretization
sofi                    ;discretization method [fofi|sofi|crni]
1  1.e+10            ;no of timesteps,size of timestep
### heat production by dissipation
n
### bouyancy by temperature gradient
boussinesq                    ;[none|boussinesq|dengrad]
### fluid regions
t                       ;for each flow region: t - fluid, f - solid
### moving grids
y                       ;lmvgr
n                       ;elliptic non-orthogonal, falls y 2 Zeilen einkommentieren
;6                      ;number of blocks
;3 4 5 6 7 8            ;block numbers in increasing order
;y                      ;elliptic orthogonal, falls y 2 Zeilen einkommentieren
;6                      ;number of orthogonal-blocks
;3 4 5 6 7 8            ;block numbers in increasing order!!
### fluid structure interaction
y  y  y                 ;lfsi  lfsiread  lfsiwrite
n                       ;estimation of distortions
50                      ;max. number of outer iterations
2                       ;coupling interface (0=none, 1=GRISSLi, 2=MpCCI)
1                       ;interpolation scheme (0=NNB, 1=CONS)
0.5 0.0                 ;undrel. param for struct distortions, aitken damping fac
0.1 0.1 0.1             ;fsi-residuum limit (x,y,z-direction)
n                       ;fsi predictor
1.e-5                   ;convergence criterion for the TFSI
```

```
1.0                     ;underrelaxation parameter for the TFSI
0                       ;Number of blocks to distort
;1 1 1 0 0 0 0 0 0
### mpcci input data
1                       ;mycode
2                       ;remoteCodeId
10                      ;meshId                  from *.cci file
4                       ;nQuantityIds
11                      ;QuantityIds(1),localMeshIds(1)
12                      ;QuantityIds(2),localMeshIds(2)
13                      ;QuantityIds(3),localMeshIds(3)
14                      ;QuantityIds(4),localMeshIds(4)
### time limit
1.e30                   ;cpu-time in seconds
### pressure gradient
0. 0. 0.                ;pressure gradient in x,y,z direction
### pressure correction steps
1                       ;feature not yet implemented !!!
### tasks per process
1                    ;for each process (processor)
### convective exit boundary condition
0                       ;switch
0.7                     ;underrelaxation for boundary condition
### turbulence statistics
none                    ;average methode [none|plane|line|time]
f                       ;higher order statistics (skewness,flatness)
100 10                  ;starting time (timestep), step (timesteps)
(500 3)                 ;starting time, step
                        ;(if higher order statistics are '.true.')
;### residuum norming
;0.02935d0 2570.d0 0.1d0 0.1d0 0.1d0 1710.d0 1.d0 ;area,rho,vel,tui,len,temp,conc
```

Listing A.1: FASTEST id-file for the tubeflow with diffuser

B FEAP Input File for the Incident Flow Simulation

The FEAP input file of the incident flow simulation.

```
FEAP
0 0 0 3 4 8

SNODes
  1 0 0 0
  2 0.02625 0 0
  3 0 0.02625 0
  4 −0.02625 0 0
  5 0 −0.02625 0
  6 0.013125 0 0
  7 0 0.013125 0
  8 −0.013125 0 0
  9 0 −0.013125 0
  10 0 0 0.00525
  11 0.02625 0 0.00525
  12 0 0.02625 0.00525
  13 −0.02625 0 0.00525
  14 0 −0.02625 0.00525
  15 0.013125 0 0.00525
  16 0 0.013125 0.00525
  17 −0.013125 0 0.00525
  18 0 −0.013125 0.00525

SIDE
  POLAr 2 5 1
  POLAr 5 4 1
  POLAr 4 3 1
  POLAr 3 2 1
  POLAr 6 9 1
  POLAr 9 8 1
  POLAr 8 7 1
  POLAr 7 6 1
  POLAr 11 14 10
  POLAr 14 13 10
  POLAr 13 12 10
  POLAr 12 11 10
  POLAr 15 18 10
  POLAr 18 17 10
  POLAr 17 16 10
  POLAr 16 15 10

BLENd 1
  SOLId 5 5 15 0 0 1
    6 2 3 7 15 11 12 16

BLENd 2
  SOLId 5 5 15 0 0 1
      4 8 7 3 13 17 16 12

BLENd 3
  SOLId 5 5 15 0 0 1
```

```
      5 9 8 4 14 18 17 13

BLENd 4
  SOLId 5 5 15 0 0 1
      5 2 6 9 14 11 15 18

BLENd 5
  SOLId 5 5 15 0 0 1
      6 7 8 9 15 16 17 18

BLENd 6
  SURFace 5 5 0 0 2
    6 7 8 9

BLENd 7
  SURFace 5 5 0 0 2
    15 16 17 18

BLENd 8
  SURFace 5 5 0 0 2
    6 2 3 7

BLENd 9
  SURFace 5 5 0 0 2
    15 11 12 16

BLENd 10
  SURFace 5 5 0 0 2
    4 8 7 3

BLENd 11
  SURFace 5 5 0 0 2
    13 17 16 12

BLENd 12
  SURFace 5 5 0 0 2
    4 5 9 8

BLENd 13
  SURFace 5 5 0 0 2
    13 14 18 17

BLENd 14
  SURFace 5 5 0 0 2
    5 2 6 9

BLENd 15
  SURFace 5 5 0 0 2
    14 11 15 18

BLENd 16
  SURFace 15 5 0 0 2
    2 11 12 3

BLENd 17
  SURFace 15 5 0 0 2
    3 12 13 4

```

```
110 BLENd 18
111   SURFace 15 5 0 0 2
112     4 13 14 5
113
114 BLENd 19
115   SURFace 15 5 0 0 2
116     5 14 11 2
117
118 END
119
120 TIE
121
122 MACR
123   FCBC,INIT
124   ...
125 END
126
127 POLA,0,0,0,0,0,0.00525,0.02625,3
128 PLAN,0,0,0,0.02625,0,0,0,0.02625,0
129 PLAN,0,0,0.00525,0.02625,0,0.00525,0,0.02625,0.00525
```

Listing B.1: FEAP Input File for the Incident Flow on a Plate

C Sourcecode of usermacro 6

Source code of the usermacro for plane, polar and coed coupling.

```
c$Id: umacr6.F,v 1.0.0.0 2008-09-16 15:43:36 nlinder Exp $
      subroutine umacr6(lct,ctl,prt)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-1998: Robert L. Taylor
c....  meynen   24.02.03

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose:  MpCCI commands for initialization, termination and
c                data exchange (send/recv).

c      Inputs:
c         lct       - Command character parameters
c         ctl(10)   - Command numerical parameters
c         prt       - Flag, output if true

c      Outputs:
c         ---
c-----[--.----+----.----+----.-----------------------------------------]

      implicit  none

      include  'iofile.h'
      include  'cdata.h'
      include  'sdata.h'
      include  'umac1.h'
      include  'pointer.h'
      include  'upointer.h'

      include  'comblk.h'

      include  'mpcci.h'

      integer   iError, i, dir
      logical   pcomp,prt,prth, pinput, errck, tinput
      logical   setvar,ualloc
      integer   nNodesPerElem
      parameter(nNodesPerElem = 4)
      character lct*15, yyy*15
      real*8    ctl(10), axis, td(12)
      save

      prth = .true.

      if(pcomp(uct,'mac6',4)) then
        uct = 'fcbc'

      elseif(pcomp(lct,'init',4)) then

         setvar = ualloc(2, 'CCI_C', numnp, 1) ! scratch
```

```
52          setvar = ualloc(4, 'CCI_H', numnp*nNodesPerElem, 1) ! scratch
53          setvar = ualloc(5, 'CCI_M', numnp, 1) ! MAP FEAP -> CCI nodes
54
55          errck = tinput(yyy,1,td,12)
56   123    if(.not. errck) then
57             errck = tinput(yyy,1,td,12)
58             if(pcomp(yyy,'plan',4)) then
59                write(*,*) "Planar coupling"
60                call defineSurfaceCouplingBC(mr(np(33)), hr(np(43)),
61       $             td(1), td(2), td(3), td(4), td(5),
62       $             td(6), td(7), td(8), td(9),
63       $             mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)
64                write(*,*) "nNodes: ", nNodes, " - nElems: ", nElems
65                goto 123
66             elseif(pcomp(yyy,'pola',4)) then
67                write(*,*) "Polar coupling"
68                call definePolarCouplingBC(mr(np(33)), hr(np(43)), td(1),
69       $             td(2), td(3), td(4), td(5), td(6), td(7), td(8),
70       $             mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)
71                write(*,*) "nNodes: ", nNodes, " - nElems: ", nElems
72                goto 123
73             elseif(pcomp(yyy,'diff',4)) then
74                write(*,*) "Coned coupling"
75                call defineDiffusorCouplingBC(mr(np(33)), hr(np(43)),
76       $             td(1),td(2),td(3),td(4), td(5), td(6), td(7), td(8),
77       $             td(9), mr(up(2)),mr(up(4)),mr(up(5)),nNodes, nElems)
78                write(*,*) "nNodes: ", nNodes, " - nElems: ", nElems
79                goto 123
80             elseif(pcomp(yyy,'0',4)) then
81                goto 345
82             endif
83   345    endif
84
85   c     optimize the alocated space
86          setvar = ualloc(1, 'CCI_N', nNodes, 1) ! cci_node
87          call pmovei(mr(up(2)), mr(up(1)), nNodes) ! CCI_C --> CCI_N
88          setvar = ualloc(2, 'CCI_C', 0, 1) ! free scratch
89          setvar = ualloc(3, 'CCI_E', nNodesPerElem*nElems, 1)
90          call pmovei(mr(up(4)), mr(up(3)), nNodesPerElem*nElems)
91   c     ! CCI_H --> CCI_E
92          setvar = ualloc(4, 'CCI_H', 0, 1) ! free scratch
93
94          endif
95
96          return
97          end
```

Listing C.1: umacr6.F

D Sourcecode of usermacro 7

Source code of the usermacro for TFI coupling.

```
1  c$Id: umacr7.F,v 1.0.0.0 2008-09-16 15:43:36 nlinder Exp $
2        subroutine umacr7(lct,ctl,prt)
3
4  c      * * F E A P * * A Finite Element Analysis Program
5
6  c....  Copyright (c) 1984-1998: Robert L. Taylor
7  c....  meynen   24.02.03
8
9  c-----[--.----+----.----+----.-----------------------------------------]
10 c      Purpose:  MpCCI commands for initialization, termination and
11 c                data exchange (send/recv).
12
13 c      Inputs:
14 c         lct       - Command character parameters
15 c         ctl(10)   - Command numerical parameters
16 c         prt       - Flag, output if true
17
18 c      Outputs:
19 c         ---
20 c-----[--.----+----.----+----.-----------------------------------------]
21
22       implicit  none
23
24       include  'iofile.h'
25       include  'cdata.h'
26       include  'sdata.h'
27       include  'umac1.h'
28       include  'pointer.h'
29       include  'upointer.h'
30
31       include  'comblk.h'
32
33       include  'mpcci.h'
34
35       integer   iError, i, dir, scratch, temp,j,ii, offset
36       logical   pcomp,prt,prth, pinput, errck, tinput
37       logical   setvar,ualloc
38       integer   nNodesPerElem
39       parameter(nNodesPerElem = 4, scratch = 10)
40       character lct*15, yyy*15
41       real*8    ctl(10), td(12), nodesco(scratch,3)
42       real*8    sppoints(scratch*10, 4), tempfirst, templast, tempnum
43       real*8    tempj, initialbend, finalbend, direqu, dirval
44       real*8    x, y, z
45       integer   tempnn(12), dis1, dis2, surfacenumber
46       save
47
48       do i = 1, scratch*10
49          do j = 1, 3
50             sppoints(i,j) = 0
51          enddo
```

```
52        enddo
53        do i = 1, 12
54           tempnn(i) = 2
55        enddo
56
57        prth = .true.
58
59        if(pcomp(uct,'mac7',4)) then
60          uct = 'tfic'
61
62        elseif(pcomp(lct,'init',4)) then
63
64           setvar = ualloc(2, 'CCI_C', numnp, 1) ! scratch
65           setvar = ualloc(4, 'CCI_H', numnp*nNodesPerElem, 1) ! scratch
66           setvar = ualloc(5, 'CCI_M', numnp, 1) ! MAP FEAP -> CCI nodes
67
68
69           errck = tinput(yyy,1,td,12)
70  123      if(.not. errck) then
71              errck = tinput(yyy,1,td,12)
72              write(*,*) yyy
73              if(pcomp(yyy,'node',4)) then
74                 temp = td(1)
75                 do i = 1, temp
76                    errck = pinput(td,4)
77                    nodesco(i,1) = td(2)
78                    nodesco(i,2) = td(3)
79                    nodesco(i,3) = td(4)
80                 enddo
81                 goto 123
82              elseif(pcomp(yyy,'curv',4)) then
83                 temp = td(1)
84                 dis1 = td(2)
85                 dis2 = td(3)
86                 surfacenumber = td(4)
87                 tempj = 1
88                 do i = 1, temp
89                    errck = pinput(td,12)
90                    tempfirst = td(2) ! first node of the curve
91                    templast = td(3) ! last node if the curve
92                    tempnum = td(5) ! number of support points
93                    initialbend = td(6)
94                    finalbend = td(7)
95                    direqu = td(8)
96                    dirval = td(9)
97                    tempnn(td(1)) = td(4)+1 ! number of nodes for i-th curve
98                    if (td(4) .eq. 1) then ! linear
99                       write(*,*) "linear spline def"
100                      sppoints(tempj,1) = nodesco(tempfirst,1)
101                      sppoints(tempj,2) = nodesco(tempfirst,2)
102                      sppoints(tempj,3) = nodesco(tempfirst,3)
103                      sppoints(tempj+1,1) = nodesco(templast,1)
104                      sppoints(tempj+1,2) = nodesco(templast,2)
105                      sppoints(tempj+1,3) = nodesco(templast,3)
106                      tempj = tempj+2
107                   endif
108
109                   if (td(4) .eq. 3) then ! cubic Spline
```

```
                    write(*,*) "cubic spline definition"
                    sppoints(tempj,1) = nodesco(tempfirst,1)
                    sppoints(tempj,2) = nodesco(tempfirst,2)
                    sppoints(tempj,3) = nodesco(tempfirst,3)
                    sppoints(tempj,4) = initialbend
                    do j = tempj+1, tempj+td(4)
                       errck = pinput(td,3)
                       sppoints(j,1) = td(1)
                       sppoints(j,2) = td(2)
                       sppoints(j,3) = td(3)
                    enddo
                    sppoints(tempj+td(4)+1,1)=nodesco(templast,1)
                    sppoints(tempj+td(4)+1,2)=nodesco(templast,2)
                    sppoints(tempj+td(4)+1,3)=nodesco(templast,3)
                    sppoints(tempj+td(4)+1,4) = finalbend
                    tempj = tempj + td(4) + 2
                 endif
              enddo
              goto 123
           elseif(pcomp(yyy,'0',4)) then
              goto 345
           endif
345     endif

        setvar = ualloc(103, 'nump', 12, 2)

        do i = 1,12
           hr(up(103)+i-1) = tempnn(i)
           if(tempnn(i) .gt. 2) tempnn(i) = tempnn(i)+1
        enddo

        call alloctfiarray(tempnn)

        offset = 0

        do i = 1,12
           if (tempnn(i) .gt. 3) then ! spline
              call columnmove(sppoints,1,hr(up(3*(i+1)+1)),
     $             tempnn(i),offset)
              call columnmove(sppoints,2,hr(up(3*(i+1)+2)),
     $             tempnn(i),offset)
              call columnmove(sppoints,3,hr(up(3*(i+1)+3)),
     $             tempnn(i),offset)
              call defineSpline(hr(up(3*(i+1)+1)),hr(up(3*(i+1)+2)),
     $             hr(up(3*(i+1)+3)), sppoints(i,4),
     $             sppoints(tempnn(i),4), tempnn(i), surfacenumber,
     $             hr(up(4*(i+9)+3)), hr(up(4*(i+9)+4)),
     $             hr(up(4*(i+9)+5)), hr(up(4*(i+9)+6)))
           elseif (tempnn(i) .eq. 2) then ! linear
              call columnmove(sppoints,1,hr(up(3*(i+1)+1)),
     $             tempnn(i),offset)
              call columnmove(sppoints,2,hr(up(3*(i+1)+2)),
     $             tempnn(i),offset)
              call columnmove(sppoints,3,hr(up(3*(i+1)+3)),
     $             tempnn(i),offset)
           endif
           offset = offset + tempnn(i)
        enddo
```

```
168
169         setvar = ualloc(104, 'tfisurfx', dis1*dis2, 2)
170         setvar = ualloc(105, 'tfisurfy', dis1*dis2, 2)
171         setvar = ualloc(106, 'tfisurfz', dis1*dis2, 2)
172
173
174 c     now TFI
175         do i = 1, dis1
176            do j = 1, dis2
177               call TFI((i-1)*(1d0/(dis1-1)),(j-1)*(1d0/(dis2-1)),
178      $                 1d0,x,y,z)
179               hr(up(104)+(i-1)*dis1+j-1) = x
180               hr(up(105)+(i-1)*dis1+j-1) = y
181               hr(up(106)+(i-1)*dis1+j-1) = z
182            enddo
183         enddo
184
185 c     Now search the coupling nodes!
186
187         call definetfisurface(mr(np(33)), hr(np(43)), dis1, dis2,
188      $        mr(up(2)),mr(up(4)),mr(up(5)),nNodes, nElems)
189
190 c     optimize the alocated space
191         setvar = ualloc(1, 'CCI_N', nNodes, 1) ! cci_node
192         call pmovei(mr(up(2)), mr(up(1)), nNodes) ! CCI_C --> CCI_N
193         setvar = ualloc(2, 'CCI_C', 0, 1) ! free scratch
194         setvar = ualloc(3, 'CCI_E', nNodesPerElem*nElems, 1)
195         call pmovei(mr(up(4)), mr(up(3)), nNodesPerElem*nElems)
196 c     ! CCI_H --> CCI_E
197         setvar = ualloc(4, 'CCI_H', 0, 1) ! free scratch
198         call cleartfiarray()
199
200         write(*,*) "nNodes: ", nNodes, " - nElems: ", nElems
201
202       endif
203
204       return
205       end
206
207       subroutine columnmove(Orig, column, Dest, length, offset)
208
209       integer  column, length, offset, i
210       real*8   Orig(100,3), Dest(length)
211
212       do i = 1, length
213          Dest(i) = Orig(i+offset,column)
214          if (length .gt. 3) then
215             write(*,*) Dest(i), i+offset, column
216          endif
217       enddo
218
219       return
220       end
```

Listing D.1: umacr7.F

E Sourcecode of boundary8brickelem

Source code of functions for finding the orientation of an eight brick element.

```
      subroutine boundary8BrickElem(cciMap, cciElems,
     $     nNodes, nElems, orientation, source)
      implicit  none
c      include  'cdata.h'
c      include  'sdata.h'

      integer cciMap(*), cciElems(*), source(*)
      integer nNodes, nElems, orientation
      integer addCouplingNode

c     local variables
      integer index, temp
      save

      index = nElems*4

      if(orientation .eq. 1) then ! west coupling face
         temp = addCouplingNode(cciMap, source(1), nNodes)
         cciElems(index+1) = temp
         temp = addCouplingNode(cciMap, source(4), nNodes)
         cciElems(index+2) = temp
         temp = addCouplingNode(cciMap, source(8), nNodes)
         cciElems(index+3) = temp
         temp = addCouplingNode(cciMap, source(5), nNodes)
         cciElems(index+4) = temp
      else if(orientation .eq. 2) then ! south
         temp = addCouplingNode(cciMap, source(1), nNodes)
         cciElems(index+1) = temp
         temp = addCouplingNode(cciMap, source(2), nNodes)
         cciElems(index+2) = temp
         temp = addCouplingNode(cciMap, source(6), nNodes)
         cciElems(index+3) = temp
         temp = addCouplingNode(cciMap, source(5), nNodes)
         cciElems(index+4) = temp
      else if(orientation .eq. 3) then ! bottom
         temp = addCouplingNode(cciMap, source(1), nNodes)
         cciElems(index+1) = temp
         temp = addCouplingNode(cciMap, source(2), nNodes)
         cciElems(index+2) = temp
         temp = addCouplingNode(cciMap, source(3), nNodes)
         cciElems(index+3) = temp
         temp = addCouplingNode(cciMap, source(4), nNodes)
         cciElems(index+4) = temp
      else if(orientation .eq. 4) then ! top
         temp = addCouplingNode(cciMap, source(5), nNodes)
         cciElems(index+1) = temp
         temp = addCouplingNode(cciMap, source(6), nNodes)
         cciElems(index+2) = temp
         temp = addCouplingNode(cciMap, source(7), nNodes)
         cciElems(index+3) = temp
         temp = addCouplingNode(cciMap, source(8), nNodes)
```

```
52          cciElems(index+4) = temp
53        else if(orientation .eq. 5) then ! north
54          temp = addCouplingNode(cciMap, source(4), nNodes)
55          cciElems(index+1) = temp
56          temp = addCouplingNode(cciMap, source(3), nNodes)
57          cciElems(index+2) = temp
58          temp = addCouplingNode(cciMap, source(7), nNodes)
59          cciElems(index+3) = temp
60          temp = addCouplingNode(cciMap, source(8), nNodes)
61          cciElems(index+4) = temp
62        else if(orientation .eq. 6) then ! east
63          temp = addCouplingNode(cciMap, source(2), nNodes)
64          cciElems(index+1) = temp
65          temp = addCouplingNode(cciMap, source(3), nNodes)
66          cciElems(index+2) = temp
67          temp = addCouplingNode(cciMap, source(7), nNodes)
68          cciElems(index+3) = temp
69          temp = addCouplingNode(cciMap, source(6), nNodes)
70          cciElems(index+4) = temp
71        endif
72        return
73        end
74
75        function addCouplingNode(cciMap, nodeindex, nNodes)
76        implicit  none
77        include  'cdata.h'
78        include  'sdata.h'
79
80        integer addCouplingNode
81        integer cciMap(*), nodeindex, nNodes
82
83        integer i
84        save
85
86        if(cciMap(nodeindex) .eq. 0) then
87          nNodes = nNodes + 1
88          cciMap(nodeindex) = nNodes
89        endif
90        addCouplingNode = cciMap(nodeindex)
91
92        return
93        end
94
95        integer function findOrientation_8Brick(elementnodes,
96       $     surfacenodes)
97        implicit none
98
99        integer elementnodes(*), surfacenodes(*), i
100       logical contains
101
102   c    orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
103       if(contains(surfacenodes, elementnodes(1), 4) .and.
104      $     contains(surfacenodes, elementnodes(4), 4) .and.
105      $     contains(surfacenodes, elementnodes(8), 4) .and.
106      $     contains(surfacenodes, elementnodes(5), 4)) then
107         findOrientation_8Brick = 1
108       else if(contains(surfacenodes, elementnodes(1), 4) .and.
109      $        contains(surfacenodes, elementnodes(2), 4) .and.
```

```
110       $        contains(surfacenodes, elementnodes(6), 4) .and.
111       $        contains(surfacenodes, elementnodes(5), 4)) then
112          findOrientation_8Brick = 2
113        else if(contains(surfacenodes, elementnodes(1), 4) .and.
114       $        contains(surfacenodes, elementnodes(2), 4) .and.
115       $        contains(surfacenodes, elementnodes(3), 4) .and.
116       $        contains(surfacenodes, elementnodes(4), 4)) then
117          findOrientation_8Brick = 3
118        else if(contains(surfacenodes, elementnodes(5), 4) .and.
119       $        contains(surfacenodes, elementnodes(6), 4) .and.
120       $        contains(surfacenodes, elementnodes(7), 4) .and.
121       $        contains(surfacenodes, elementnodes(8), 4)) then
122          findOrientation_8Brick = 4
123        else if(contains(surfacenodes, elementnodes(4), 4) .and.
124       $        contains(surfacenodes, elementnodes(3), 4) .and.
125       $        contains(surfacenodes, elementnodes(7), 4) .and.
126       $        contains(surfacenodes, elementnodes(8), 4)) then
127          findOrientation_8Brick = 5
128        else if(contains(surfacenodes, elementnodes(2), 4) .and.
129       $        contains(surfacenodes, elementnodes(3), 4) .and.
130       $        contains(surfacenodes, elementnodes(7), 4) .and.
131       $        contains(surfacenodes, elementnodes(6), 4)) then
132          findOrientation_8Brick = 6
133        endif
134
135        return
136        end
137
138        logical function contains(source, element, n)
139        implicit none
140
141        integer source(*), element, n
142        integer i
143
144        contains = .false.
145        do i = 1, n
146           if(source(i) .eq. element) then
147              contains = .true.
148              goto 111
149           endif
150    111    continue
151        enddo
152
153        return
154        end
```

Listing E.1: boundary8brickelem.F

F Sourcecode of definesurfcbc

Source code of the functions for finding coupling nodes on plane surfaces.

```
      subroutine defineSurfaceCouplingBC(ix, x, x1, y1, z1, x2, y2,
     $     z2, x3, y3, z3, cciNodes, cciElems, cciMap, nNodes, nElems)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-2004: Regents of the University of California
c                               All rights reserved

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose: Define an coupling BC for rectangular surfaces.
c      Inputs:
c         ix          - the element connection array in feap
c         x           - the coordinates array in feap
c         xn, yn, zn  - The 3 nodes defining the Surface where the
c                       Boundary Condition should be applied
c      Outputs:
c         cciNodes    - Array with all nodes which lay on the boundary
c         cciElems    - Array with all elements which lay on the boundary
c         cciMap      - Array containing the connection FEAP index-> cci index
c(the opposite of cciNodes)
c         nNodes      - number of interface nodes on the defined boundary
c         nElems      - number of interface elements on the interface boundary
c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none
      include  'cdata.h'
      include  'sdata.h'
c      include  'iofile.h'

      integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
      integer nNodes, nElems
      real*8  x(ndm, *)
      real*8  x1,x2,x3,y1,y2,y3,z1,z2,z3

c     local variables
      real*8  Vertex(3,3), n(3), b, TesVer(3)
      real*8  temp, dif, dis
      integer nNodesPerElem
      parameter (dif = 1.d-10, nnodesPerElem = 4)
      integer i, j, k, count, scount, orientation, axis
c     orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
      integer couplelem(nen), surfaceelem(nen)
      integer findOrientation_8Brick
      logical flag
      save

c     Vertex should be an array with the coordinates of the three
c     nodes defining the rectangular contact surface
c     HNF will compute the normal vector stored in n
c     b is the scaling factor (computed in HNF)

      Vertex(1,1) = x1
```

```
52        Vertex(2,1) = y1
53        Vertex(3,1) = z1
54        Vertex(1,2) = x2
55        Vertex(2,2) = y2
56        Vertex(3,2) = z2
57        Vertex(1,3) = x3
58        Vertex(2,3) = y3
59        Vertex(3,3) = z3
60
61        call HNF(Vertex,n,b)
62
63        do i = 1, numel
64           flag = .false.
65           count  = 0
66           scount = 0
67           do j = 1, nen
68              if(ix(j, i) .eq. 0) then ! surface element
69                 flag = .false.
70              else
71                 count = count + 1
72                 couplelem(count) = ix(j, i)
73
74  c     dis_Ve_Pl computes the distance between the vertex TesVer and the
75  c     coupling surface and is stored in dis
76  c     TesVer is an array with the coordinates of the Vertex j in Element i
77
78                 TesVer(1) = x(1,ix(j,i))
79                 TesVer(2) = x(2,ix(j,i))
80                 TesVer(3) = x(3,ix(j,i))
81
82                 call dis_Ve_Pl(b,n,TesVer,Vertex,dis)
83
84                 if(dis .lt. dif) then
85                    flag = .true.
86                    scount = scount + 1
87                    surfaceelem(scount) = ix(j, i)
88                 endif
89              endif
90           enddo
91           axis = 1
92           if(flag) then          ! coupling boundary element
93              orientation = findOrientation_8Brick(couplelem,
94       $           surfaceelem)
95              call boundary8BrickElem(cciMap, cciElems,
96       $           nNodes, nElems, orientation, couplelem)
97
98              nElems = nElems + 1
99           endif
100       enddo
101
102 c     now we have the filled cciMap. Construct the cciNodes array
103       do i = 1, numnp
104          if(cciMap(i) .ne. 0) then ! coupling node
105             cciNodes(cciMap(i)) = i
106          endif
107       enddo
108
109       return
```

```
110       end
111
112       subroutine HNF(Vertex,n,b)
113
114       implicit none
115       real*8 Vertex(3,*), n(*), b
116
117 c     local variables:
118       real*8 u(3), v(3)
119
120       u(1) = Vertex(1,1) - Vertex(1,2)
121       u(2) = Vertex(2,1) - Vertex(2,2)
122       u(3) = Vertex(3,1) - Vertex(3,2)
123
124       v(1) = Vertex(1,1) - Vertex(1,3)
125       v(2) = Vertex(2,1) - Vertex(2,3)
126       v(3) = Vertex(3,1) - Vertex(3,3)
127
128       n(1) = u(2)*v(3) - u(3)*v(2)
129       n(2) = u(3)*v(1) - u(1)*v(3)
130       n(3) = u(1)*v(2) - u(2)*v(1)
131
132       b = sqrt(n(1)*n(1) + n(2)*n(2) + n(3)*n(3))
133
134       return
135       end
136
137
138       subroutine dis_Ve_Pl(b,n,TesVer, Vertex,dis)
139
140       implicit none
141       real*8 TesVer(*), n(*), b, Vertex(3,*), dis
142
143
144       dis = dabs((TesVer(1)-Vertex(1,3))*n(1) +
145      $     (TesVer(2)-Vertex(2,3))*n(2) +
146      $     (TesVer(3)-Vertex(3,3))*n(3))
147       dis = dis/b
148
149       return
150       end
```

Listing F.1: definesurfcbc.F

G Sourcecode of definepolcbc

Source code of the functions for finding coupling nodes on polar surfaces.

```
      subroutine definePolarCouplingBC(ix, x, x1, y1, z1, x2, y2, z2, r,
     $     or, cciNodes, cciElems, cciMap, nNodes, nElems)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-2004: Regents of the University of California
c                               All rights reserved

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose: Define an coupling BC for polar surfaces (tubes,..).
c      Inputs:
c         ix          - the element connection array in feap
c         x           - the coordinates array in feap
c         x1, y1, z1
c         x2, y2, z2  - The 2 nodes defining the axis of the tube
c         r           - The radius
c         or          - The orientation of the axis (1, 2, or 3)
c
c      Outputs:
c         cciNodes    - Array with all nodes which lay on the boundary
c         cciElems    - Array with all elements which lay on the boundary
c         cciMap      - Array containing the connection FEAP index-> cci index
c(the opposite of cciNodes)
c         nNodes      - number of interface nodes on the defined boundary
c         nElems      - number of interface elements on the interface boundary
c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none
      include  'cdata.h'
      include  'sdata.h'
c      include  'iofile.h'

      real*8  x(ndm, *)
      integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
      integer nNodes, nElems
      real*8 x1, y1, z1, x2, y2, z2, r, or

c     local variables
      real*8  g(3), dis, P(3), Ax(3,2), dif, temp, disnodes(8)
      integer nNodesPerElem
      parameter (dif = 1.d-8, nnodesPerElem = 4)
      integer i, j, k,l, count, scount, orientation, axis, ornodes(4)
c     orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
      integer couplelem(nen), surfaceelem(nen)
      integer findOrientation_8Brick, ppp
      logical flag, contains, onax
      save

c     Organizing the input data
      Ax(1,1) = x1
      Ax(2,1) = y1
      Ax(3,1) = z1
```

```
52        Ax(1,2) = x2
53        Ax(2,2) = y2
54        Ax(3,2) = z2
55
56  c     computaing the direction vector (g)
57        call LinEq(Ax,g)
58
59        do i = 1, numel
60           flag = .false.
61           onax = .true.
62           count  = 0
63           scount = 0
64           ornodes(1) = 0
65           ornodes(2) = 0
66           ornodes(3) = 0
67           ornodes(4) = 0
68
69           do ppp = 1, 8
70              disnodes(ppp) = 0
71              surfaceelem(ppp) = 0
72           enddo
73
74           do j = 1, nen
75              if(ix(j, i) .eq. 0) then ! surface element
76                 flag = .false.
77              else
78                 count = count + 1
79                 couplelem(count) = ix(j, i)
80
81                 P(1) = x(1,ix(j,i))
82                 P(2) = x(2,ix(j,i))
83                 P(3) = x(3,ix(j,i))
84
85  c              Check if the current node is outside the Axis
86                 if (P(or) .gt. Ax(or,2)) then
87                    onax = .false.
88                 elseif (P(or) .lt. Ax(or,1)) then
89                    onax = .false.
90                 endif
91
92                 if (.not. onax) then
93                    flag = .false.
94                    goto 767
95
96                 endif
97
98  c              Computing the distance of the Point to the Axis (dis)
99                 call dis_Ve_Ax(g,Ax,P,dis)
100
101                temp = dabs(dis - r)
102                disnodes(j) = temp
103
104                if(temp .lt. dif) then
105                   flag = .true.
106                   scount = scount + 1
107                   surfaceelem(scount) = ix(j, i)
108                   write(*,*) P(1), P(2), P(3), scount
109                endif
```

```
110             endif
111          enddo
112  767     continue
113
114          if(flag) then          ! coupling boundary element
115             orientation = findOrientation_8Brick(couplelem,
116     $           surfaceelem)
117             call boundary8BrickElem(cciMap, cciElems,
118     $           nNodes, nElems, orientation, couplelem)
119             nElems = nElems + 1
120          endif
121       enddo
122
123 c     now we have the filled cciMap. Construct the cciNodes array
124       do i = 1, numnp
125          if(cciMap(i) .ne. 0) then ! coupling node
126             cciNodes(cciMap(i)) = i
127          endif
128       enddo
129
130       return
131       end
132
133       subroutine LinEq(Ax,g)
134 c     Subroutine to compute the direction vector
135
136       implicit none
137       real*8 Ax(3,*), g(*)
138       real*8 u(3), v(3)
139
140       g(1) = Ax(1,1)-Ax(1,2)
141       g(2) = Ax(2,1)-Ax(2,2)
142       g(3) = Ax(3,1)-Ax(3,2)
143
144       return
145       end
146
147
148       subroutine dis_Ve_Ax(g,Ax,P,dis)
149 c     subroutine to compute the distance of Point P the the Axis (Ax, g)
150       implicit none
151
152       real*8 g(*), Ax(3,*), P(*), dis, d(3)
153
154       dis=(g(1)*(P(1)-Ax(1,1))+g(2)*(P(2)-Ax(2,1))+g(3)*(P(3)-Ax(3,1)))/
155      $     (g(1)*g(1)+g(2)*g(2)+g(3)*g(3));
156
157       d(1)=Ax(1,1)+dis*g(1)-P(1);
158       d(2)=Ax(2,1)+dis*g(2)-P(2);
159       d(3)=Ax(3,1)+dis*g(3)-P(3);
160
161       dis=sqrt(d(1)*d(1)+d(2)*d(2)+d(3)*d(3));
162
163       return
164       end
```

Listing G.1: definepolcbc.F

H Sourcecode of definediffcbc

Source code of the functions for finding coupling nodes on coned surfaces.

```
      subroutine defineDiffusorCouplingBC(ix, x, x1, y1, z1, x2, y2, z2,
     $     dirr, r1, r2, cciNodes, cciElems, cciMap, nNodes, nElems)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-2004: Regents of the University of California
c                               All rights reserved

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose: Define an coupling BC for diffusors.
c      Inputs:
c         ix          - the element connection array in feap
c         x           - the coordinates array in feap
c         x1, x2      - The 2 nodes defining the axis
c         r1          - The smaller radius
c         r2          - The greater radius
c         dir         - 1, 2 or 3 -> Direction of the diffusor
c
c      Outputs:
c         cciNodes    - Array with all nodes which lay on the boundary
c         cciElems    - Array with all elements which lay on the boundary
c         cciMap      - Array containing the connection FEAP index-> cci index
c(the opposite of cciNodes)
c         nNodes      - number of interface nodes on the defined boundary
c         nElems      - number of interface elements on the interface boundary
c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none
      include  'cdata.h'
      include  'sdata.h'
c      include  'iofile.h'

      real*8  x(ndm, *)
      integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
      integer nNodes, nElems
      real*8 x1, y1, z1, x2, y2, z2, r1, r2

c     local variables
      real*8  g(3), P(3), Ax(3,2), temp, DP(3,2), set, gv(3)
      real*8 dis1, dis2, dif, L, dirr, disnodes(8)
      integer nNodesPerElem, nn
      parameter (dif = 1.d-8, nnodesPerElem = 4)
      integer i, j, k, m, count, scount, orientation, axis, ornodes(4)
c     orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
      integer couplelem(nen), surfaceelem(nen)
      integer findOrientation_8Brick, dir, ppp
      logical flag, contains, onax
      save

c     Organizing the input data
      if (dirr .eq. 1) then
         dir = 1
```

```
52        elseif (dirr .eq. 2) then
53           dir = 2
54        elseif (dirr .eq. 3) then
55           dir = 3
56        endif
57
58        write(*,*) dir, dirr
59
60        do nn = 1, 3
61           g(nn) = 0
62           gv(nn) = 0
63        enddo
64
65        Ax(1,1) = x1
66        Ax(2,1) = y1
67        Ax(3,1) = z1
68        Ax(1,2) = x2
69        Ax(2,2) = y2
70        Ax(3,2) = z2
71
72        DP(1,1) = Ax(1,1)
73        DP(2,1) = Ax(2,1)
74        DP(3,1) = Ax(3,1)
75        DP(1,2) = Ax(1,2)
76        DP(2,2) = Ax(2,2)
77        DP(3,2) = Ax(3,2)
78
79        DP(2,1) = DP(2,1)+r1
80        DP(2,2) = DP(2,2)+r2
81
82        L = dabs(Ax(dir,2) - Ax(dir,1))
83
84  c     Computing the drection vectors of the Axis and the diffusor
85        call LinEq2(Ax,g,dir)
86
87        call LinEq2(DP,gv,dir)
88
89        do i = 1, numel
90           flag = .false.
91           count  = 0
92           scount = 0
93           ornodes(1) = 0
94           ornodes(2) = 0
95           ornodes(3) = 0
96           ornodes(4) = 0
97           onax = .true.
98           do ppp = 1, 8
99              disnodes(ppp) = 0
100             surfaceelem(ppp) = 0
101          enddo
102
103          do j = 1, nen
104             if(ix(j, i) .eq. 0) then ! surface element
105                flag = .false.
106             else
107                count = count + 1
108                couplelem(count) = ix(j, i)
109
```

```
110               P(1) = x(1,ix(j,i))
111               P(2) = x(2,ix(j,i))
112               P(3) = x(3,ix(j,i))
113
114               if (P(dir) .gt. Ax(dir,2)) then
115                  onax = .false.
116               elseif (P(dir) .lt. Ax(dir,1)) then
117                  onax = .false.
118               endif
119
120               if (.not. onax) then
121                  flag = .false.
122                  goto 767
123               endif
124
125 c             computing the distance of the current Vertex to the axis and
126 c             the distance a point must have to be on the diffusor at this
127 c             point
128               call dis_Ve_Ax2(g,Ax,P,dis1)
129
130               call dif_rad(P,set,Ax,gv,dir,r1,L)
131
132               temp = dabs(dis1 - set)
133               disnodes(j) = temp
134
135               if(temp .lt. dif) then
136                  flag = .true.
137                  scount = scount + 1
138                  surfaceelem(scount) = ix(j, i)
139               endif
140            endif
141         enddo
142  767    continue
143
144         if(flag) then          ! coupling boundary element
145
146            orientation = findOrientation_8Brick(couplelem,
147      $           surfaceelem)
148            call boundary8BrickElem(cciMap, cciElems,
149      $           nNodes, nElems, orientation, couplelem)
150
151            nElems = nElems + 1
152         endif
153       enddo
154
155 c     now we have the filled cciMap. Construct the cciNodes array
156       do i = 1, numnp
157          if(cciMap(i) .ne. 0) then ! coupling node
158             cciNodes(cciMap(i)) = i
159          endif
160       enddo
161
162       return
163       end
164
165       subroutine LinEq2(Te,t,dir)
166
167 c     computing the direction vector
```

```
168
169       implicit none
170       real*8 Te(3,*), t(*)
171       integer dir
172
173       t(dir) = Te(dir,2)-Te(dir,1)
174
175       return
176       end
177
178       subroutine dis_Ve_Ax2(g,Ax,P,dis)
179
180 c     computing the distance of point P to Axis (g, Ax)
181
182       implicit none
183       real*8 g(*), Ax(3,*), P(*), dis, d(3)
184
185
186       dis=(g(1)*(P(1)-Ax(1,1))+g(2)*(P(2)-Ax(2,1))+g(3)*(P(3)-Ax(3,1)))/
187      $     (g(1)*g(1)+g(2)*g(2)+g(3)*g(3));
188
189       d(1)=Ax(1,1)+dis*g(1)-P(1);
190       d(2)=Ax(2,1)+dis*g(2)-P(2);
191       d(3)=Ax(3,1)+dis*g(3)-P(3);
192
193       dis=sqrt(d(1)*d(1)+d(2)*d(2)+d(3)*d(3));
194
195       return
196       end
197
198 ccccccccccccccccccccccccc
199 c     Soubroutine which computes the radius of the volute
200 c     at a given point on the axis
201 ccccccccccccccccccccccccc
202
203       subroutine dif_rad(P,set,Ax,gv,dir,r1,L)
204
205       implicit none
206       real*8 P(*), set, Ax(3,*), gv(*), r1, L, temp
207       integer dir
208
209       if (Ax(dir,1) .gt. Ax(dir,2)) then
210          temp = Ax(dir,2)
211       else
212          temp = Ax(dir,1)
213       endif
214
215       set = P(dir) - temp
216       set = set*gv(dir)/L+r1
217
218       return
219       end
```

Listing H.1: definediffcbc.F

I Sourcecode of definetfisurface

Source code of the functions for finding coupling nodes on surfaces discribed by the use of TFI.

```
      subroutine definetfisurface(ix, x, dis1, dis2,
     $     cciNodes, cciElems, cciMap, nNodes, nElems)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-2004: Regents of the University of California
c                               All rights reserved

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose: Define an coupling BC for rectangular surfaces.
c      Inputs:
c         ix          - the element connection array in feap
c         x           - the coordinates array in feap
c         dis1,dis2   - the discrtisation of the grid
c         all other paramters are taken from the arrays
c
c      Outputs:
c         cciNodes    - Array with all nodes which lay on the boundary
c         cciElems    - Array with all elements which lay on the boundary
c         cciMap      - Array containing the connection FEAP index-> cci index
c(the opposite of cciNodes)
c         nNodes      - number of interface nodes on the defined boundary
c         nElems      - number of interface elements on the interface boundary
c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none
      include  'cdata.h'
      include  'sdata.h'

      include  'pointer.h'
      include  'upointer.h'

      include  'comblk.h'

      real*8  x(ndm, *)
      integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
      integer nNodes, nElems, dis1, dis2

c     local variables
      real*8  P(3), temp, dif, distance
      integer nNodesPerElem
      parameter (dif = 1.d-3, nnodesPerElem = 4)
      integer i, j, count, scount, orientation, axis
c     orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
      integer couplelem(nen), surfaceelem(nen)
      integer findOrientation_8Brick, ppp,lll
      logical flag
      save

      do i = 1, numel
         flag = .false.
         count  = 0
```

```
52          scount = 0
53
54          do j = 1, nen
55             if(ix(j, i) .eq. 0) then ! surface element
56                flag = .false.
57             else
58                count = count + 1
59                couplelem(count) = ix(j, i)
60
61                P(1) = x(1,ix(j,i))
62                P(2) = x(2,ix(j,i))
63                P(3) = x(3,ix(j,i))
64
65                call isonsurf(P, distance, dis1, dis2)
66
67                if(distance .lt. dif) then
68                   flag = .true.
69                   scount = scount + 1
70                   surfaceelem(scount) = ix(j, i)
71                endif
72             endif
73          enddo
74
75          if(flag) then          ! coupling boundary element
76             orientation = findOrientation_8Brick(couplelem,
77      $           surfaceelem)
78             call boundary8BrickElem(cciMap, cciElems,
79      $           nNodes, nElems, orientation, couplelem)
80
81             nElems = nElems + 1
82          endif
83       enddo
84
85 c     now we have the filled cciMap. Construct the cciNodes array
86       do i = 1, numnp
87          if(cciMap(i) .ne. 0) then ! coupling node
88             cciNodes(cciMap(i)) = i
89          endif
90       enddo
91
92       return
93       end
94
95       subroutine isonsurf(P, distance, dis1, dis2)
96 c     computes the distance of P to the TFI-surface
97       implicit none
98
99       include  'pointer.h'
100      include  'upointer.h'
101
102      include  'comblk.h'
103
104      real*8  P(*), temp, temp1, temp2, temp3, distance
105      integer dis1, dis2, i, j
106
107
108      distance = 10d10
109
```

```
110         do i = 1, dis1
111            do j = 1, dis2
112               temp1 = P(1) - hr(up(104)+(i-1)*dis1+j-1)
113               temp2 = P(2) - hr(up(105)+(i-1)*dis1+j-1)
114               temp3 = P(3) - hr(up(106)+(i-1)*dis1+j-1)
115               temp = temp1*temp1 + temp2*temp2 + temp3*temp3
116               temp = sqrt(temp)
117
118               if (temp .lt. distance) distance = temp
119
120            enddo
121         enddo
122
123         return
124         end
```

Listing I.1: definetfisurface.F

J Sourcecode of TFI

Source code of the functions transferring from logical coordinates to real coordinates by the use of TFI.

```
      subroutine TFI(xi, eta, zeta, x, y, z)
      implicit none

      real*8 xi, eta, zeta, x, y, z
      real*8 ux, uy, uz, vx, vy, vz, wx, wy, wz
      real*8 uwx, uwy, uwz, uvx, uvy, uvz, vwx, vwy, vwz
      real*8 uvwx, uvwy, uvwz
      real*8 f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12
      real*8 d0, d1

      parameter (d0 = 0.d0, d1 = 1.d0)
c     Goal: Computing all coordinates of the surface

      ux = (1-xi)*((1-zeta)*f5(d0,eta,d0,1)+zeta*f8(d0,eta,d1,1)
     &     +(1-eta)*f9(d0,d0,zeta,1)+eta*f12(d0,d1,zeta,1)
     &     -eta*(zeta*f8(d0,d1,d1,1)+(1-zeta)*f5(d0,d1,d0,1))
     &     -(1-eta)*(zeta*f8(d0,d0,d1,1)+(1-zeta)*f1(d0,d0,d0,1)))
     &     +xi*((1-zeta)*f6(d1,eta,d0,1)+zeta*f7(d1,eta,d1,1)
     &     +(1-eta)*f10(d1,d0,zeta,1)+eta*f11(d1,d1,zeta,1)
     &     -eta*(zeta*f7(d1,d1,d1,1)+(1-zeta)*f6(d1,d1,d0,1))
     &     -(1-eta)*(zeta*f7(d1,d0,d1,1)+(1-zeta)*f1(d1,d0,d0,1)))

      uy = (1-xi)*((1-zeta)*f5(d0,eta,d0,2)+zeta*f8(d0,eta,d1,2)
     &     +(1-eta)*f9(d0,d0,zeta,2)+eta*f12(d0,d1,zeta,2)
     &     -eta*(zeta*f8(d0,d1,d1,2)+(1-zeta)*f5(d0,d1,d0,2))
     &     -(1-eta)*(zeta*f8(d0,d0,d1,2)+(1-zeta)*f1(d0,d0,d0,2)))
     &     +xi*((1-zeta)*f6(d1,eta,d0,2)+zeta*f7(d1,eta,d1,2)
     &     +(1-eta)*f10(d1,d0,zeta,2)+eta*f11(d1,d1,zeta,2)
     &     -eta*(zeta*f7(d1,d1,d1,2)+(1-zeta)*f6(d1,d1,d0,2))
     &     -(1-eta)*(zeta*f7(d1,d0,d1,2)+(1-zeta)*f1(d1,d0,d0,2)))

      uz = (1-xi)*((1-zeta)*f5(d0,eta,d0,3)+zeta*f8(d0,eta,d1,3)
     &     +(1-eta)*f9(d0,d0,zeta,3)+eta*f12(d0,d1,zeta,3)
     &     -eta*(zeta*f8(d0,d1,d1,3)+(1-zeta)*f5(d0,d1,d0,3))
     &     -(1-eta)*(zeta*f8(d0,d0,d1,3)+(1-zeta)*f1(d0,d0,d0,3)))
     &     +xi*((1-zeta)*f6(d1,eta,d0,3)+zeta*f7(d1,eta,d1,3)
     &     +(1-eta)*f10(d1,d0,zeta,3)+eta*f11(d1,d1,zeta,3)
     &     -eta*(zeta*f7(d1,d1,d1,3)+(1-zeta)*f6(d1,d1,d0,3))
     &     -(1-eta)*(zeta*f7(d1,d0,d1,3)+(1-zeta)*f1(d1,d0,d0,3)))

      vx = (1-eta)*((1-zeta)*f1(xi,d0,d0,1)+zeta*f2(xi,d0,d1,1)
     &     +(1-xi)*f9(d0,d0,zeta,1)+xi*f10(d1,d0,zeta,1)
     &     -xi*(zeta*f2(d1,d0,d1,1)+(1-zeta)*f1(d1,d0,d0,1))
     &     -(1-xi)*(zeta*f9(d0,d0,d1,1)+(1-zeta)*f1(d0,d0,d0,1)))
     &     +eta*((1-zeta)*f4(xi,d1,d0,1)+zeta*f3(xi,d1,d1,1)
     &     +(1-xi)*f12(d0,d1,zeta,1)+xi*f11(d1,d1,zeta,1)
     &     -xi*(zeta*f7(d1,d1,d1,1)+(1-zeta)*f6(d1,d1,d0,1))
     &     -(1-xi)*(zeta*f8(d0,d1,d1,1)+(1-zeta)*f5(d0,d1,d0,1)))

      vy = (1-eta)*((1-zeta)*f1(xi,d0,d0,2)+zeta*f2(xi,d0,d1,2)
     &     +(1-xi)*f9(d0,d0,zeta,2)+xi*f10(d1,d0,zeta,2)
```

```
52        &      -xi*(zeta*f2(d1,d0,d1,2)+(1-zeta)*f1(d1,d0,d0,2))
53        &      -(1-xi)*(zeta*f9(d0,d0,d1,2)+(1-zeta)*f1(d0,d0,d0,2)))
54        &      +eta*((1-zeta)*f4(xi,d1,d0,2)+zeta*f3(xi,d1,d1,2)
55        &      +(1-xi)*f12(d0,d1,zeta,2)+xi*f11(d1,d1,zeta,2)
56        &      -xi*(zeta*f7(d1,d1,d1,2)+(1-zeta)*f6(d1,d1,d0,2))
57        &      -(1-xi)*(zeta*f8(d0,d1,d1,2)+(1-zeta)*f5(d0,d1,d0,2)))
58
59         vz = (1-eta)*((1-zeta)*f1(xi,d0,d0,3)+zeta*f2(xi,d0,d1,3)
60        &      +(1-xi)*f9(d0,d0,zeta,3)+xi*f10(d1,d0,zeta,3)
61        &      -xi*(zeta*f2(d1,d0,d1,3)+(1-zeta)*f1(d1,d0,d0,3))
62        &      -(1-xi)*(zeta*f9(d0,d0,d1,3)+(1-zeta)*f1(d0,d0,d0,3)))
63        &      +eta*((1-zeta)*f4(xi,d1,d0,3)+zeta*f3(xi,d1,d1,3)
64        &      +(1-xi)*f12(d0,d1,zeta,3)+xi*f11(d1,d1,zeta,3)
65        &      -xi*(zeta*f7(d1,d1,d1,3)+(1-zeta)*f6(d1,d1,d0,3))
66        &      -(1-xi)*(zeta*f8(d0,d1,d1,3)+(1-zeta)*f5(d0,d1,d0,3)))
67
68         wx = (1-zeta)*((1-eta)*f1(xi,d0,d0,1)+eta*f4(xi,d1,d0,1)
69        &      +(1-xi)*f5(d0,eta,d0,1)+xi*f6(d1,eta,d0,1)
70        &      -xi*(eta*f6(d1,d1,d0,1)+(1-eta)*f1(d1,d0,d0,1))
71        &      -(1-xi)*(eta*f5(d0,d1,d0,1)+(1-eta)*f1(d0,d0,d0,1)))
72        &      +zeta*((1-eta)*f2(xi,d0,d1,1)+eta*f3(xi,d1,d1,1)
73        &      +(1-xi)*f8(d0,eta,d1,1)+xi*f7(d1,eta,d1,1)
74        &      -xi*(eta*f7(d1,d1,d1,1)+(1-eta)*f2(d1,d0,d1,1))
75        &      -(1-xi)*(eta*f12(d0,d1,d1,1)+(1-eta)*f9(d0,d0,d1,1)))
76
77         wy = (1-zeta)*((1-eta)*f1(xi,d0,d0,2)+eta*f4(xi,d1,d0,2)
78        &      +(1-xi)*f5(d0,eta,d0,2)+xi*f6(d1,eta,d0,2)
79        &      -xi*(eta*f6(d1,d1,d0,2)+(1-eta)*f1(d1,d0,d0,2))
80        &      -(1-xi)*(eta*f5(d0,d1,d0,2)+(1-eta)*f1(d0,d0,d0,2)))
81        &      +zeta*((1-eta)*f2(xi,d0,d1,2)+eta*f3(xi,d1,d1,2)
82        &      +(1-xi)*f8(d0,eta,d1,2)+xi*f7(d1,eta,d1,2)
83        &      -xi*(eta*f7(d1,d1,d1,2)+(1-eta)*f2(d1,d0,d1,2))
84        &      -(1-xi)*(eta*f12(d0,d1,d1,2)+(1-eta)*f9(d0,d0,d1,2)))
85
86         wz = (1-zeta)*((1-eta)*f1(xi,d0,d0,3)+eta*f4(xi,d1,d0,3)
87        &      +(1-xi)*f5(d0,eta,d0,3)+xi*f6(d1,eta,d0,3)
88        &      -xi*(eta*f6(d1,d1,d0,3)+(1-eta)*f1(d1,d0,d0,3))
89        &      -(1-xi)*(eta*f5(d0,d1,d0,3)+(1-eta)*f1(d0,d0,d0,3)))
90        &      +zeta*((1-eta)*f2(xi,d0,d1,3)+eta*f3(xi,d1,d1,3)
91        &      +(1-xi)*f8(d0,eta,d1,3)+xi*f7(d1,eta,d1,3)
92        &      -xi*(eta*f7(d1,d1,d1,3)+(1-eta)*f2(d1,d0,d1,3))
93        &      -(1-xi)*(eta*f12(d0,d1,d1,3)+(1-eta)*f9(d0,d0,d1,3)))
94
95         uwx=(1-xi)*(1-zeta)*f5(d0,eta,d0,1)+(1-xi)*zeta*
96        &      f8(d0,eta,d1,1)+xi*(1-zeta)*f6(d1,eta,d0,1)+
97        &      xi*zeta*f7(d1,eta,d1,1)
98
99         uwy=(1-xi)*(1-zeta)*f5(d0,eta,d0,2)+(1-xi)*zeta*
100       &      f8(d0,eta,d1,2)+xi*(1-zeta)*f6(d1,eta,d0,2)+
101       &      xi*zeta*f7(d1,eta,d1,2)
102
103        uwz=(1-xi)*(1-zeta)*f5(d0,eta,d0,3)+(1-xi)*zeta*
104       &      f8(d0,eta,d1,3)+xi*(1-zeta)*f6(d1,eta,d0,3)+
105       &      xi*zeta*f7(d1,eta,d1,3)
106
107        uvx=(1-xi)*(1-eta)*f9(d0,d0,zeta,1)+(1-xi)*eta*
108       &      f12(d0,d1,zeta,1)+xi*(1-eta)*f10(d1,d0,zeta,1)+
109       &      xi*eta*f11(d1,d1,zeta,1)
```

```
110
111       uvy=(1-xi)*(1-eta)*f9(d0,d0,zeta,2)+(1-xi)*eta*
112      &      f12(d0,d1,zeta,2)+xi*(1-eta)*f10(d1,d0,zeta,2)+
113      &      xi*eta*f11(d1,d1,zeta,2)
114
115       uvz=(1-xi)*(1-eta)*f9(d0,d0,zeta,3)+(1-xi)*eta*
116      &      f12(d0,d1,zeta,3)+xi*(1-eta)*f10(d1,d0,zeta,3)+
117      &      xi*eta*f11(d1,d1,zeta,3)
118
119       vwx=(1-eta)*(1-zeta)*f1(xi,d0,d0,1)+(1-zeta)*eta*
120      &      f4(xi,d1,d0,1)+zeta*(1-eta)*f2(xi,d0,d1,1)+
121      &      eta*zeta*f3(xi,1d1,1d1,1)
122
123       vwy=(1-eta)*(1-zeta)*f1(xi,d0,d0,2)+(1-zeta)*eta*
124      &      f4(xi,d1,d0,2)+zeta*(1-eta)*f2(xi,d0,d1,2)+
125      &      eta*zeta*f3(xi,d1,d1,2)
126
127       vwz=(1-eta)*(1-zeta)*f1(xi,d0,d0,3)+(1-zeta)*eta*
128      &      f4(xi,d1,d0,3)+zeta*(1-eta)*f2(xi,d0,d1,3)+
129      &      eta*zeta*f3(xi,d1,d1,3)
130
131       uvwx=(1-xi)*(1-eta)*(1-zeta)*f1(d0,d0,d0,1)+
132      &      (1-xi)*(1-eta)*zeta*f9(d0,d0,d1,1)+
133      &      (1-xi)*eta*(1-zeta)*f5(d0,d1,d0,1)+
134      &      (1-xi)*eta*zeta*f8(d0,d1,d1,1)+
135      &      xi*(1-eta)*(1-zeta)*f1(d1,d0,d0,1)+
136      &      xi*(1-eta)*zeta*f2(d1,d0,d1,1)+
137      &      xi*eta*(1-zeta)*f6(d1,d1,d0,1)+
138      &      xi*eta*zeta*f7(d1,d1,d1,1)
139
140       uvwy=(1-xi)*(1-eta)*(1-zeta)*f1(d0,d0,d0,2)+
141      &      (1-xi)*(1-eta)*zeta*f9(d0,d0,d1,2)+
142      &      (1-xi)*eta*(1-zeta)*f5(d0,d1,d0,2)+
143      &      (1-xi)*eta*zeta*f8(d0,d1,d1,2)+
144      &      xi*(1-eta)*(1-zeta)*f1(d1,d0,d0,2)+
145      &      xi*(1-eta)*zeta*f2(d1,d0,d1,2)+
146      &      xi*eta*(1-zeta)*f6(d1,d1,d0,2)+
147      &      xi*eta*zeta*f7(d1,d1,d1,2)
148
149       uvwz=(1-xi)*(1-eta)*(1-zeta)*f1(d0,d0,d0,3)+
150      &      (1-xi)*(1-eta)*zeta*f9(d0,d0,d1,3)+
151      &      (1-xi)*eta*(1-zeta)*f5(d0,d1,d0,3)+
152      &      (1-xi)*eta*zeta*f8(d0,d1,d1,3)+
153      &      xi*(1-eta)*(1-zeta)*f1(d1,d0,d0,3)+
154      &      xi*(1-eta)*zeta*f2(d1,d0,d1,3)+
155      &      xi*eta*(1-zeta)*f6(d1,d1,d0,3)+
156      &      xi*eta*zeta*f7(d1,d1,d1,3)
157
158       x = ux+vx+wx-uwx-uvx-vwx+uvwx
159       y = uy+vy+wy-uwy-uvy-vwy+uvwy
160       z = uz+vz+wz-uwz-uvz-vwz+uvwz
161
162       return
163       end
```

Listing J.1: TFI.F

K Sourcecode of functions

Source code of the functions used for the TFI.

```
c-----[--.----+----.----+----.-----------------------------------------]
c      Functions: f1 - f12
c      Purpose: Defining the function for an edge of the tfisurface.
c      Inputs:
c         xi,eta,zeta - logical coordinates
c         dir         - Direction of the edge (1, 2 or 3)
c
c      Outputs:
c         stores the paramters directly in the arrays
c-----[--.----+----.----+----.-----------------------------------------]
      function f1(xi,eta,zeta,dir)

      include  'pointer.h'
      include  'upointer.h'

      include  'comblk.h'

      real*8 f1

      real*8 xi, eta, zeta, llog, lre, xtemp, yspline
      integer n, dir, j
      real*8 P1(3), P2(3), g(3)
      real*8 x, y, z, temp
      logical flag1, flag2

      if (hr(up(103)) .eq. 2) then ! linear
         P1(1) = hr(up(7))
         P2(1) = hr(up(7)+1)
         P1(2) = hr(up(8))
         P2(2) = hr(up(8)+1)
         P1(3) = hr(up(9))
         P2(3) = hr(up(9)+1)

         g(1) = P2(1)-P1(1)
         g(2) = P2(2)-P1(2)
         g(3) = P2(3)-P1(3)

         x = P1(1) + xi*g(1)
         y = P1(2) + eta*g(2)
         z = P1(3) + zeta*g(3)

         if (dir .eq. 1) f1 = x
         if (dir .eq. 2) f1 = y
         if (dir .eq. 3) f1 = z

      elseif (hr(up(103)) .gt. 2) then ! Spline
         j = hr(up(103))
         P1(1) = hr(up(7))
         P2(1) = hr(up(7)+j)
         P1(2) = hr(up(8))
         P2(2) = hr(up(8)+j)
```

```
52          P1(3) = hr(up(9))
53          P2(3) = hr(up(9)+j)
54
55          g(1) = P2(1)-P1(1)
56          g(2) = P2(2)-P1(2)
57          g(3) = P2(3)-P1(3)
58
59          int = 0
60
61          xtemp = P1(1) + xi * g(1)
62
63          do i = 1, j ! in which splinecurve?
64             if (xtemp .ge.hr(up(7)+i-1).and.xtemp .le. hr(up(7)+i)) then
65                int = i-1
66             endif
67          enddo
68
69          yspline = hr(up(43)+int)*(xtemp - hr(up(7)+int))*
70     $         (xtemp -hr(up(7)+int))*(xtemp -hr(up(7)+int))
71     $         + hr(up(44)+int) * (xtemp -hr(up(7)+int))*
72     $         (xtemp -hr(up(7)+int))
73     $         + hr(up(45)+int) *(xtemp -hr(up(7)+int))
74     $         + hr(up(46)+int)
75
76          if (dir .eq. 1) f1 = xtemp
77          if (dir .eq. 3) f1 = P1(3) + zeta*g(3)
78          if (dir .eq. 2) f1 = yspline
79
80       endif
81
82       return
83       end
84
85       function f2(xi,eta,zeta,dir)
86
87       include  'pointer.h'
88       include  'upointer.h'
89
90       include  'comblk.h'
91
92       real*8 f2
93
94       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
95       integer n, dir, j
96       real*8 P1(3), P2(3), g(3)
97       real*8 x, y, z, temp
98       logical flag1, flag2
99
100      if (hr(up(103)+1) .eq. 2) then ! linear
101         P1(1) = hr(up(10))
102         P2(1) = hr(up(10)+1)
103         P1(2) = hr(up(11))
104         P2(2) = hr(up(11)+1)
105         P1(3) = hr(up(12))
106         P2(3) = hr(up(12)+1)
107
108         g(1) = P2(1)-P1(1)
109         g(2) = P2(2)-P1(2)
```

```
110         g(3) = P2(3)-P1(3)
111
112         x = P1(1) + xi*g(1)
113         y = P1(2) + eta*g(2)
114         z = P1(3) + zeta*g(3)
115
116         if (dir .eq. 1) f2 = x
117         if (dir .eq. 2) f2 = y
118         if (dir .eq. 3) f2 = z
119
120       elseif (hr(up(103)+1) .gt. 2) then ! Spline
121         j = hr(up(103)+1)
122         P1(1) = hr(up(10))
123         P2(1) = hr(up(10)+j)
124         P1(2) = hr(up(11))
125         P2(2) = hr(up(11)+j)
126         P1(3) = hr(up(12))
127         P2(3) = hr(up(12)+j)
128
129         g(1) = P2(1)-P1(1)
130         g(2) = P2(2)-P1(2)
131         g(3) = P2(3)-P1(3)
132
133         int = 0
134
135         xtemp = P1(1) + xi * g(1)
136
137         do i = 1, j ! in which splinecurve?
138            if (xtemp.ge.hr(up(10)+i-1).and.xtemp .le. hr(up(10)+i))then
139               int = i-1
140            endif
141         enddo
142
143         yspline = hr(up(47)+int)*(xtemp - hr(up(10)+int))*
144      $         (xtemp -hr(up(10)+int))*(xtemp -hr(up(10)+int))
145      $         + hr(up(48)+int) * (xtemp -hr(up(10)+int))*
146      $         (xtemp -hr(up(10)+int))
147      $         + hr(up(49)+int) *(xtemp -hr(up(10)+int))
148      $         + hr(up(50)+int)
149
150         if (dir .eq. 1) f2 = xtemp
151         if (dir .eq. 3) f2 = P1(3) + zeta*g(3)
152         if (dir .eq. 2) f2 = yspline
153
154
155       endif
156
157       return
158       end
159
160       function f3(xi,eta,zeta,dir)
161
162       include  'pointer.h'
163       include  'upointer.h'
164
165       include  'comblk.h'
166
167       real*8 f3
```

```
168
169       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
170       integer n, dir, j
171       real*8 P1(3), P2(3), g(3)
172       real*8 x, y, z, temp
173       logical flag1, flag2
174
175       if (hr(up(103)+2) .eq. 2) then ! linear
176          P1(1) = hr(up(13))
177          P2(1) = hr(up(13)+1)
178          P1(2) = hr(up(14))
179          P2(2) = hr(up(14)+1)
180          P1(3) = hr(up(15))
181          P2(3) = hr(up(15)+1)
182
183          g(1) = P2(1)-P1(1)
184          g(2) = P2(2)-P1(2)
185          g(3) = P2(3)-P1(3)
186
187          x = P1(1) + xi*g(1)
188          y = P1(2) + eta*g(2)
189          z = P1(3) + zeta*g(3)
190
191          if (dir .eq. 1) f3 = x
192          if (dir .eq. 2) f3 = y
193          if (dir .eq. 3) f3 = z
194
195
196       elseif (hr(up(103)+2) .gt. 1) then ! Spline
197          j = hr(up(103)+2)
198          P1(1) = hr(up(13))
199          P2(1) = hr(up(13)+j)
200          P1(2) = hr(up(14))
201          P2(2) = hr(up(14)+j)
202          P1(3) = hr(up(15))
203          P2(3) = hr(up(15)+j)
204
205          g(1) = P2(1)-P1(1)
206          g(2) = P2(2)-P1(2)
207          g(3) = P2(3)-P1(3)
208
209          int = 0
210
211          xtemp = P1(1) + xi * g(1)
212
213          do i = 1, j ! in which splinecurve?
214             if (xtemp.ge.hr(up(13)+i-1).and.xtemp .le. hr(up(13)+i))then
215                int = i-1
216             endif
217          enddo
218
219          yspline = hr(up(51)+int)*(xtemp - hr(up(13)+int))*
220       $        (xtemp -hr(up(13)+int))*(xtemp -hr(up(13)+int))
221       $        + hr(up(52)+int) * (xtemp -hr(up(13)+int))*
222       $        (xtemp -hr(up(13)+int))
223       $        + hr(up(53)+int) *(xtemp -hr(up(13)+int))
224       $        + hr(up(54)+int)
225
```

```
226         if (dir .eq. 1) f3 = xtemp
227         if (dir .eq. 3) f3 = P1(3) + zeta*g(3)
228         if (dir .eq. 2) f3 = yspline
229
230      endif
231
232      return
233      end
234
235      function f4(xi,eta,zeta,dir)
236
237      include  'pointer.h'
238      include  'upointer.h'
239
240      include  'comblk.h'
241
242      real*8 f4
243
244      real*8 xi, eta, zeta, llog, lre, xtemp, yspline
245      integer n, dir, j
246      real*8 P1(3), P2(3), g(3)
247      real*8 x, y, z, temp
248      logical flag1, flag2
249
250      if (hr(up(103)+3) .eq. 2) then ! linear
251         P1(1) = hr(up(16))
252         P2(1) = hr(up(16)+1)
253         P1(2) = hr(up(17))
254         P2(2) = hr(up(17)+1)
255         P1(3) = hr(up(18))
256         P2(3) = hr(up(18)+1)
257
258         g(1) = P2(1)-P1(1)
259         g(2) = P2(2)-P1(2)
260         g(3) = P2(3)-P1(3)
261
262         x = P1(1) + xi*g(1)
263         y = P1(2) + eta*g(2)
264         z = P1(3) + zeta*g(3)
265
266         if (dir .eq. 1) f4 = x
267         if (dir .eq. 2) f4 = y
268         if (dir .eq. 3) f4 = z
269
270      elseif (hr(up(103)+3) .gt. 2) then ! Spline
271         j = hr(up(103)+3)
272         P1(1) = hr(up(16))
273         P2(1) = hr(up(16)+j)
274         P1(2) = hr(up(17))
275         P2(2) = hr(up(17)+j)
276         P1(3) = hr(up(18))
277         P2(3) = hr(up(18)+j)
278
279         g(1) = P2(1)-P1(1)
280         g(2) = P2(2)-P1(2)
281         g(3) = P2(3)-P1(3)
282
283         int = 0
```

```
284
285         xtemp = P1(1) + xi * g(1)
286
287         do i = 1, j ! in which splinecurve?
288            if (xtemp.ge.hr(up(16)+i-1).and.xtemp .le. hr(up(16)+i))then
289               int = i-1
290            endif
291         enddo
292
293         yspline = hr(up(55)+int)*(xtemp - hr(up(16)+int))*
294      $        (xtemp -hr(up(16)+int))*(xtemp -hr(up(16)+int))
295      $        + hr(up(56)+int) * (xtemp -hr(up(16)+int))*
296      $        (xtemp -hr(up(16)+int))
297      $        + hr(up(57)+int) *(xtemp -hr(up(16)+int))
298      $        + hr(up(58)+int)
299
300         if (dir .eq. 1) f4 = xtemp
301         if (dir .eq. 3) f4 = P1(3) + zeta*g(3)
302         if (dir .eq. 2) f4 = yspline
303
304       endif
305
306       return
307       end
308
309       function f5(xi,eta,zeta,dir)
310
311       include  'pointer.h'
312       include  'upointer.h'
313
314       include  'comblk.h'
315
316       real*8 f5
317
318       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
319       integer n, dir, j
320       real*8 P1(3), P2(3), g(3)
321       real*8 x, y, z, temp
322       logical flag1, flag2
323
324       if (hr(up(103)+4) .eq. 2) then ! linear
325          P1(1) = hr(up(19))
326          P2(1) = hr(up(19)+1)
327          P1(2) = hr(up(20))
328          P2(2) = hr(up(20)+1)
329          P1(3) = hr(up(21))
330          P2(3) = hr(up(21)+1)
331
332          g(1) = P2(1)-P1(1)
333          g(2) = P2(2)-P1(2)
334          g(3) = P2(3)-P1(3)
335
336          x = P1(1) + xi*g(1)
337          y = P1(2) + eta*g(2)
338          z = P1(3) + zeta*g(3)
339
340          if (dir .eq. 1) f5 = x
341          if (dir .eq. 2) f5 = y
```

```
342           if (dir .eq. 3) f5 = z
343
344         elseif (hr(up(103)+4) .gt. 2) then ! Spline
345           j = hr(up(103)+4)
346           P1(1) = hr(up(19))
347           P2(1) = hr(up(19)+j)
348           P1(2) = hr(up(20))
349           P2(2) = hr(up(20)+j)
350           P1(3) = hr(up(21))
351           P2(3) = hr(up(21)+j)
352
353           g(1) = P1(1)-P2(1)
354           g(2) = P1(2)-P2(2)
355           g(3) = P1(3)-P2(3)
356
357           do i = 1, j
358             if (hr(up(19)) .eq. hr(up(19)+i)) flag1 = .true.
359             if (hr(up(21)) .eq. hr(up(21)+i)) flag2 = .true.
360           enddo
361
362           xtemp = P1(2) + eta * g(2)
363
364           do i = 1, j ! in which splinecurve?
365             if (xtemp.ge.hr(up(20)+i-1).and.xtemp .le. hr(up(20)+i))then
366               int = i-1
367             endif
368           enddo
369
370           yspline = hr(up(59)+int)*(xtemp - hr(up(20)+int))*
371      $          (xtemp -hr(up(20)+int))*(xtemp -hr(up(20)+int))
372      $          + hr(up(60)+int) * (xtemp -hr(up(20)+int))*
373      $          (xtemp -hr(up(20)+int))
374      $          + hr(up(61)+int) *(xtemp -hr(up(20)+int))
375      $          + hr(up(62)+int)
376
377           if (dir .eq. 2) f5 = xtemp
378           if (dir .eq. 1) f5 = P1(1) + xi*g(1)
379           if (dir .eq. 3) f5 = yspline
380
381         endif
382
383         return
384         end
385
386         function f6(xi,eta,zeta,dir)
387
388         include  'pointer.h'
389         include  'upointer.h'
390
391         include  'comblk.h'
392
393         real*8 f6
394
395         real*8 xi, eta, zeta, llog, lre, xtemp, yspline
396         integer n, dir, j
397         real*8 P1(3), P2(3), g(3)
398         real*8 x, y, z, temp
399         logical flag1, flag2
```

```
400
401       if (hr(up(103)+5) .eq. 2) then ! linear
402          P1(1) = hr(up(22))
403          P2(1) = hr(up(22)+1)
404          P1(2) = hr(up(23))
405          P2(2) = hr(up(23)+1)
406          P1(3) = hr(up(24))
407          P2(3) = hr(up(24)+1)
408
409          g(1) = P2(1)-P1(1)
410          g(2) = P2(2)-P1(2)
411          g(3) = P2(3)-P1(3)
412
413          x = P1(1) + xi*g(1)
414          y = P1(2) + eta*g(2)
415          z = P1(3) + zeta*g(3)
416
417          if (dir .eq. 1) f6 = x
418          if (dir .eq. 2) f6 = y
419          if (dir .eq. 3) f6 = z
420
421
422       elseif (hr(up(103)+5) .gt. 2) then ! Spline
423          j = hr(up(103)+5)
424          P1(1) = hr(up(22))
425          P2(1) = hr(up(22)+j)
426          P1(2) = hr(up(23))
427          P2(2) = hr(up(23)+j)
428          P1(3) = hr(up(24))
429          P2(3) = hr(up(24)+j)
430
431          g(1) = P2(1)-P1(1)
432          g(2) = P2(2)-P1(2)
433          g(3) = P2(3)-P1(3)
434
435          do i = 1, j
436             if (hr(up(22)) .eq. hr(up(22)+i)) flag1 = .true.
437             if (hr(up(24)) .eq. hr(up(24)+i)) flag2 = .true.
438          enddo
439
440          xtemp = P1(2) + eta * g(2)
441
442          do i = 1, j ! in which splinecurve?
443             if (xtemp.ge.hr(up(23)+i-1).and.xtemp .le. hr(up(23)+i))then
444                int = i-1
445             endif
446          enddo
447
448          yspline = hr(up(63)+int)*(xtemp - hr(up(23)+int))*
449       $         (xtemp -hr(up(23)+int))*(xtemp -hr(up(23)+int))
450       $         + hr(up(64)+int) * (xtemp -hr(up(23)+int))*
451       $         (xtemp -hr(up(23)+int))
452       $         + hr(up(65)+int) *(xtemp -hr(up(23)+int))
453       $         + hr(up(66)+int)
454
455          if (dir .eq. 2) f6 = xtemp
456          if (dir .eq. 1) f6 = P1(1) + xi*g(1)
457          if (dir .eq. 3) f6 = yspline
```

```
458
459
460         endif
461
462         return
463         end
464
465         function f7(xi,eta,zeta,dir)
466
467         include  'pointer.h'
468         include  'upointer.h'
469
470         include  'comblk.h'
471
472         real*8 f7
473
474         real*8 xi, eta, zeta, llog, lre, xtemp, yspline
475         integer n, dir, j
476         real*8 P1(3), P2(3), g(3)
477         real*8 x, y, z, temp
478         logical flag1, flag2
479
480         if (hr(up(103)+6) .eq. 2) then ! linear
481            P1(1) = hr(up(25))
482            P2(1) = hr(up(25)+1)
483            P1(2) = hr(up(26))
484            P2(2) = hr(up(26)+1)
485            P1(3) = hr(up(27))
486            P2(3) = hr(up(27)+1)
487
488            g(1) = P2(1)-P1(1)
489            g(2) = P2(2)-P1(2)
490            g(3) = P2(3)-P1(3)
491
492            x = P1(1) + xi*g(1)
493            y = P1(2) + eta*g(2)
494            z = P1(3) + zeta*g(3)
495
496            if (dir .eq. 1) f7 = x
497            if (dir .eq. 2) f7 = y
498            if (dir .eq. 3) f7 = z
499
500         elseif (hr(up(103)+6) .gt. 2) then ! Spline
501            j = hr(up(103)+6)
502            P1(1) = hr(up(25))
503            P2(1) = hr(up(25)+j)
504            P1(2) = hr(up(26))
505            P2(2) = hr(up(26)+j)
506            P1(3) = hr(up(27))
507            P2(3) = hr(up(27)+j)
508
509            g(1) = P2(1)-P1(1)
510            g(2) = P2(2)-P1(2)
511            g(3) = P2(3)-P1(3)
512
513            int = 0
514
515            xtemp = P1(2) + eta * g(2)
```

```
516
517         do i = 1, j ! in which splinecurve?
518            if (xtemp.ge.hr(up(26)+i-1).and.xtemp .le. hr(up(26)+i))then
519               int = i-1
520            endif
521         enddo
522
523         yspline = hr(up(67)+int)*(xtemp - hr(up(26)+int))*
524     $        (xtemp -hr(up(26)+int))*(xtemp -hr(up(26)+int))
525     $        + hr(up(68)+int) * (xtemp -hr(up(26)+int))*
526     $        (xtemp -hr(up(26)+int))
527     $        + hr(up(69)+int) *(xtemp -hr(up(26)+int))
528     $        + hr(up(70)+int)
529
530         if (dir .eq. 2) f7 = xtemp
531         if (dir .eq. 1) f7 = P1(1) + xi*g(1)
532         if (dir .eq. 3) f7 = yspline
533
534       endif
535
536       return
537       end
538
539       function f8(xi,eta,zeta,dir)
540
541       include  'pointer.h'
542       include  'upointer.h'
543
544       include  'comblk.h'
545
546       real*8 f8
547
548       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
549       integer n, dir, j
550       real*8 P1(3), P2(3), g(3)
551       real*8 x, y, z, temp
552       logical flag1, flag2
553
554       flag1 = .false.
555       flag2 = .false.
556
557       if (hr(up(103)+7) .eq. 2) then ! linear
558          P1(1) = hr(up(28))
559          P2(1) = hr(up(28)+1)
560          P1(2) = hr(up(29))
561          P2(2) = hr(up(29)+1)
562          P1(3) = hr(up(30))
563          P2(3) = hr(up(30)+1)
564
565          g(1) = P2(1)-P1(1)
566          g(2) = P2(2)-P1(2)
567          g(3) = P2(3)-P1(3)
568
569          x = P1(1) + xi*g(1)
570          y = P1(2) + eta*g(2)
571          z = P1(3) + zeta*g(3)
572
573          if (dir .eq. 1) f8 = x
```

```
574           if (dir .eq. 2) f8 = y
575           if (dir .eq. 3) f8 = z
576
577         elseif (hr(up(103)+7) .gt. 2) then ! Spline
578           j = hr(up(103)+7)
579           P1(1) = hr(up(28))
580           P2(1) = hr(up(28)+j)
581           P1(2) = hr(up(29))
582           P2(2) = hr(up(29)+j)
583           P1(3) = hr(up(30))
584           P2(3) = hr(up(30)+j)
585
586           g(1) = P2(1)-P1(1)
587           g(2) = P2(2)-P1(2)
588           g(3) = P2(3)-P1(3)
589
590           xtemp = P1(2) + eta * g(2)
591
592           int = 0
593
594           do i = 1, j ! in which splinecurve?
595             if (xtemp.ge.hr(up(29)+i-1).and.xtemp .le. hr(up(29)+i))then
596               int = i-1
597             endif
598           enddo
599
600           yspline = hr(up(71)+int)*(xtemp - hr(up(29)+int))*
601      $          (xtemp -hr(up(29)+int))*(xtemp -hr(up(29)+int))
602      $          + hr(up(72)+int) * (xtemp -hr(up(29)+int))*
603      $          (xtemp -hr(up(29)+int))
604      $          + hr(up(73)+int) *(xtemp -hr(up(29)+int))
605      $          + hr(up(74)+int)
606
607           if (dir .eq. 2) f8 = xtemp
608           if (dir .eq. 1) f8 = P1(1) + xi*g(1)
609           if (dir .eq. 3) f8 = yspline
610
611         endif
612
613         return
614         end
615
616
617         function f9(xi,eta,zeta,dir)
618
619         include  'pointer.h'
620         include  'upointer.h'
621
622         include  'comblk.h'
623
624         real*8 f9
625
626         real*8 xi, eta, zeta, llog, lre, xtemp, yspline
627         integer n, dir, j
628         real*8 P1(3), P2(3), g(3)
629         real*8 x, y, z, temp
630         logical flag1, flag2
631
```

```
632       if (hr(up(103)+8) .eq. 2) then ! linear
633          P1(1) = hr(up(31))
634          P2(1) = hr(up(31)+1)
635          P1(2) = hr(up(32))
636          P2(2) = hr(up(32)+1)
637          P1(3) = hr(up(33))
638          P2(3) = hr(up(33)+1)
639
640          g(1) = P2(1)-P1(1)
641          g(2) = P2(2)-P1(2)
642          g(3) = P2(3)-P1(3)
643
644          x = P1(1) + xi*g(1)
645          y = P1(2) + eta*g(2)
646          z = P1(3) + zeta*g(3)
647
648          if (dir .eq. 1) f9 = x
649          if (dir .eq. 2) f9 = y
650          if (dir .eq. 3) f9 = z
651
652       elseif (hr(up(103)+8) .gt. 2) then ! Spline
653          j = hr(up(103)+8)
654          P1(1) = hr(up(31))
655          P2(1) = hr(up(31)+j)
656          P1(2) = hr(up(32))
657          P2(2) = hr(up(32)+j)
658          P1(3) = hr(up(33))
659          P2(3) = hr(up(33)+j)
660
661          g(1) = P2(1)-P1(1)
662          g(2) = P2(2)-P1(2)
663          g(3) = P2(3)-P1(3)
664
665          xtemp = P1(3) + zeta * g(3)
666
667          int = 0
668
669          do i = 1, j ! in which splinecurve?
670             if (xtemp.ge.hr(up(33)+i-1).and.xtemp .le. hr(up(33)+i))then
671                int = i-1
672             endif
673          enddo
674
675          yspline = hr(up(75)+int)*(xtemp - hr(up(33)+int))*
676      $         (xtemp -hr(up(33)+int))*(xtemp -hr(up(33)+int))
677      $         + hr(up(76)+int) * (xtemp -hr(up(33)+int))*
678      $         (xtemp -hr(up(33)+int))
679      $         + hr(up(77)+int) *(xtemp -hr(up(33)+int))
680      $         + hr(up(78)+int)
681
682          if (dir .eq. 3) f9 = xtemp
683          if (dir .eq. 2) f9 = P1(2) + eta*g(2)
684          if (dir .eq. 1) f9 = yspline
685
686       endif
687
688       return
689       end
```

```
690
691       function f10(xi,eta,zeta,dir)
692
693       include  'pointer.h'
694       include  'upointer.h'
695
696       include  'comblk.h'
697
698       real*8 f10
699
700       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
701       integer n, dir, j
702       real*8 P1(3), P2(3), g(3)
703       real*8 x, y, z, temp
704       logical flag1, flag2
705
706       if (hr(up(103)+9) .eq. 2) then ! linear
707          P1(1) = hr(up(34))
708          P2(1) = hr(up(34)+1)
709          P1(2) = hr(up(35))
710          P2(2) = hr(up(35)+1)
711          P1(3) = hr(up(36))
712          P2(3) = hr(up(36)+1)
713
714          g(1) = P2(1)-P1(1)
715          g(2) = P2(2)-P1(2)
716          g(3) = P2(3)-P1(3)
717
718          x = P1(1) + xi*g(1)
719          y = P1(2) + eta*g(2)
720          z = P1(3) + zeta*g(3)
721
722          if (dir .eq. 1) f10 = x
723          if (dir .eq. 2) f10 = y
724          if (dir .eq. 3) f10 = z
725
726       elseif (hr(up(103)+9) .gt. 2) then ! Spline
727          j = hr(up(103)+9)
728          P1(1) = hr(up(34))
729          P2(1) = hr(up(34)+j)
730          P1(2) = hr(up(35))
731          P2(2) = hr(up(35)+j)
732          P1(3) = hr(up(36))
733          P2(3) = hr(up(36)+j)
734
735          g(1) = P2(1)-P1(1)
736          g(2) = P2(2)-P1(2)
737          g(3) = P2(3)-P1(3)
738
739          int = 0
740
741          xtemp = P1(3) + zeta * g(3)
742
743          do i = 1, j ! in which splinecurve?
744             if (xtemp.ge.hr(up(36)+i-1).and.xtemp .le. hr(up(36)+i))then
745                int = i-1
746             endif
747          enddo
```

```
748
749         yspline = hr(up(79)+int)*(xtemp - hr(up(36)+int))*
750      $        (xtemp -hr(up(36)+int))*(xtemp -hr(up(36)+int))
751      $        + hr(up(80)+int) * (xtemp -hr(up(36)+int))*
752      $        (xtemp -hr(up(36)+int))
753      $        + hr(up(81)+int) *(xtemp -hr(up(36)+int))
754      $        + hr(up(82)+int)
755
756         if (dir .eq. 3) f10 = xtemp
757         if (dir .eq. 2) f10 = P1(2) + eta*g(2)
758         if (dir .eq. 1) f10 = yspline
759
760       endif
761
762       return
763       end
764
765       function f11(xi,eta,zeta,dir)
766
767       include  'pointer.h'
768       include  'upointer.h'
769
770       include  'comblk.h'
771
772       real*8 f11
773
774       real*8 xi, eta, zeta, llog, lre, xtemp, yspline
775       integer n, dir, j
776       real*8 P1(3), P2(3), g(3)
777       real*8 x, y, z, temp
778       logical flag1, flag2
779
780       if (hr(up(103)+10) .eq. 2) then ! linear
781          P1(1) = hr(up(37))
782          P2(1) = hr(up(37)+1)
783          P1(2) = hr(up(38))
784          P2(2) = hr(up(38)+1)
785          P1(3) = hr(up(39))
786          P2(3) = hr(up(39)+1)
787
788          g(1) = P2(1)-P1(1)
789          g(2) = P2(2)-P1(2)
790          g(3) = P2(3)-P1(3)
791
792          x = P1(1) + xi*g(1)
793          y = P1(2) + eta*g(2)
794          z = P1(3) + zeta*g(3)
795
796          if (dir .eq. 1) f11 = x
797          if (dir .eq. 2) f11 = y
798          if (dir .eq. 3) f11 = z
799
800       elseif (hr(up(103)+10) .gt. 2) then ! Spline
801          j = hr(up(103)+10)
802          P1(1) = hr(up(37))
803          P2(1) = hr(up(37)+j)
804          P1(2) = hr(up(38))
805          P2(2) = hr(up(38)+j)
```

```
806           P1(3) = hr(up(39))
807           P2(3) = hr(up(39)+j)
808
809           g(1) = P2(1)-P1(1)
810           g(2) = P2(2)-P1(2)
811           g(3) = P2(3)-P1(3)
812
813           int = 0
814
815           xtemp = P1(3) + zeta * g(3)
816
817           do i = 1, j ! in which splinecurve?
818              if (xtemp.ge.hr(up(39)+i-1).and.xtemp .le. hr(up(39)+i))then
819                 int = i-1
820              endif
821           enddo
822
823           yspline = hr(up(83)+int)*(xtemp - hr(up(39)+int))*
824          $         (xtemp -hr(up(39)+int))*(xtemp -hr(up(39)+int))
825          $         + hr(up(84)+int) * (xtemp -hr(up(39)+int))*
826          $         (xtemp -hr(up(39)+int))
827          $         + hr(up(85)+int) *(xtemp -hr(up(39)+int))
828          $         + hr(up(86)+int)
829
830           if (dir .eq. 3) f11 = xtemp
831           if (dir .eq. 2) f11 = P1(2) + eta*g(2)
832           if (dir .eq. 1) f11 = yspline
833
834        endif
835
836        return
837        end
838
839        function f12(xi,eta,zeta,dir)
840
841        include  'pointer.h'
842        include  'upointer.h'
843
844        include  'comblk.h'
845
846        real*8 f12
847
848        real*8 xi, eta, zeta, llog, lre, xtemp, yspline
849        integer n, dir, j
850        real*8 P1(3), P2(3), g(3)
851        real*8 x, y, z, temp
852        logical flag1, flag2
853
854        if (hr(up(103)+11) .eq. 2) then ! linear
855           P1(1) = hr(up(40))
856           P2(1) = hr(up(40)+1)
857           P1(2) = hr(up(41))
858           P2(2) = hr(up(41)+1)
859           P1(3) = hr(up(42))
860           P2(3) = hr(up(42)+1)
861
862           g(1) = P2(1)-P1(1)
863           g(2) = P2(2)-P1(2)
```

```
864           g(3) = P2(3)-P1(3)
865
866           x = P1(1) + xi*g(1)
867           y = P1(2) + eta*g(2)
868           z = P1(3) + zeta*g(3)
869
870           if (dir .eq. 1) f12 = x
871           if (dir .eq. 2) f12 = y
872           if (dir .eq. 3) f12 = z
873
874         elseif (hr(up(103)+11) .gt. 2) then ! Spline
875           j = hr(up(103)+11)
876           P1(1) = hr(up(40))
877           P2(1) = hr(up(40)+j)
878           P1(2) = hr(up(41))
879           P2(2) = hr(up(41)+j)
880           P1(3) = hr(up(42))
881           P2(3) = hr(up(42)+j)
882
883           g(1) = P2(1)-P1(1)
884           g(2) = P2(2)-P1(2)
885           g(3) = P2(3)-P1(3)
886
887           int = 0
888
889           xtemp = P1(3) + zeta * g(3)
890
891           do i = 1, j ! in which splinecurve?
892             if (xtemp.ge.hr(up(42)+i-1).and.xtemp .le. hr(up(42)+i))then
893               int = i-1
894             endif
895           enddo
896
897           yspline = hr(up(87)+int)*(xtemp - hr(up(42)+int))*
898       $         (xtemp -hr(up(42)+int))*(xtemp -hr(up(42)+int))
899       $         + hr(up(88)+int) * (xtemp -hr(up(42)+int))*
900       $         (xtemp -hr(up(42)+int))
901       $         + hr(up(89)+int) *(xtemp -hr(up(42)+int))
902       $         + hr(up(90)+int)
903
904           if (dir .eq. 3) f12 = xtemp
905           if (dir .eq. 2) f12 = P1(2) + eta*g(2)
906           if (dir .eq. 1) f12 = yspline
907
908         endif
909
910         return
911         end
```

Listing K.1: functions.F

L Sourcecode of alloctfiarray

Function allocating the arrays user for TFI.

```
1       subroutine alloctfiarray(tempnn)
2
3  c    Subroutine to allocate all needed arrays for the TFI
4
5       implicit none
6       integer tempnn(12)
7       logical   setvar,ualloc
8
9       setvar = ualloc(43, 'A1', tempnn(1), 2)
10      setvar = ualloc(44, 'B1', tempnn(1), 2)
11      setvar = ualloc(45, 'C1', tempnn(1), 2)
12      setvar = ualloc(46, 'D1', tempnn(1), 2)
13      setvar = ualloc(47, 'A2', tempnn(2), 2)
14      setvar = ualloc(48, 'B2', tempnn(2), 2)
15      setvar = ualloc(49, 'C2', tempnn(2), 2)
16      setvar = ualloc(50, 'D2', tempnn(2), 2)
17      setvar = ualloc(51, 'A3', tempnn(3), 2)
18      setvar = ualloc(52, 'B3', tempnn(3), 2)
19      setvar = ualloc(53, 'C3', tempnn(3), 2)
20      setvar = ualloc(54, 'D3', tempnn(3), 2)
21      setvar = ualloc(55, 'A4', tempnn(4), 2)
22      setvar = ualloc(56, 'B4', tempnn(4), 2)
23      setvar = ualloc(57, 'C4', tempnn(4), 2)
24      setvar = ualloc(58, 'D4', tempnn(4), 2)
25      setvar = ualloc(59, 'A5', tempnn(5), 2)
26      setvar = ualloc(60, 'B5', tempnn(5), 2)
27      setvar = ualloc(61, 'C5', tempnn(5), 2)
28      setvar = ualloc(62, 'D5', tempnn(5), 2)
29      setvar = ualloc(63, 'A6', tempnn(6), 2)
30      setvar = ualloc(64, 'B6', tempnn(6), 2)
31      setvar = ualloc(65, 'C6', tempnn(6), 2)
32      setvar = ualloc(66, 'D6', tempnn(6), 2)
33      setvar = ualloc(67, 'A7', tempnn(7), 2)
34      setvar = ualloc(68, 'B7', tempnn(7), 2)
35      setvar = ualloc(69, 'C7', tempnn(7), 2)
36      setvar = ualloc(70, 'D7', tempnn(7), 2)
37      setvar = ualloc(71, 'A8', tempnn(8), 2)
38      setvar = ualloc(72, 'B8', tempnn(8), 2)
39      setvar = ualloc(73, 'C8', tempnn(8), 2)
40      setvar = ualloc(74, 'D8', tempnn(8), 2)
41      setvar = ualloc(75, 'A9', tempnn(9), 2)
42      setvar = ualloc(76, 'B9', tempnn(9), 2)
43      setvar = ualloc(77, 'C9', tempnn(9), 2)
44      setvar = ualloc(78, 'D9', tempnn(9), 2)
45      setvar = ualloc(79, 'A10', tempnn(10), 2)
46      setvar = ualloc(80, 'B10', tempnn(10), 2)
47      setvar = ualloc(81, 'C10', tempnn(10), 2)
48      setvar = ualloc(82, 'D10', tempnn(10), 2)
49      setvar = ualloc(83, 'A11', tempnn(11), 2)
50      setvar = ualloc(84, 'B11', tempnn(11), 2)
51      setvar = ualloc(85, 'C11', tempnn(11), 2)
```

```
52        setvar = ualloc(86, 'D11', tempnn(11), 2)
53        setvar = ualloc(87, 'A12', tempnn(12), 2)
54        setvar = ualloc(88, 'B12', tempnn(12), 2)
55        setvar = ualloc(89, 'C12', tempnn(12), 2)
56        setvar = ualloc(90, 'D12', tempnn(12), 2)
57
58        setvar = ualloc(7, 'TFI1_N1', tempnn(1), 2)
59        setvar = ualloc(8, 'TFI1_N2', tempnn(1), 2)
60        setvar = ualloc(9, 'TFI1_N3', tempnn(1), 2)
61        setvar = ualloc(10, 'TFI2_N1', tempnn(2), 2)
62        setvar = ualloc(11, 'TFI2_N2', tempnn(2), 2)
63        setvar = ualloc(12, 'TFI2_N3', tempnn(2), 2)
64        setvar = ualloc(13, 'TFI3_N1', tempnn(3), 2)
65        setvar = ualloc(14, 'TFI3_N2', tempnn(3), 2)
66        setvar = ualloc(15, 'TFI3_N3', tempnn(3), 2)
67        setvar = ualloc(16, 'TFI4_N1', tempnn(4), 2)
68        setvar = ualloc(17, 'TFI4_N2', tempnn(4), 2)
69        setvar = ualloc(18, 'TFI4_N3', tempnn(4), 2)
70        setvar = ualloc(19, 'TFI5_N1', tempnn(5), 2)
71        setvar = ualloc(20, 'TFI5_N2', tempnn(5), 2)
72        setvar = ualloc(21, 'TFI5_N3', tempnn(5), 2)
73        setvar = ualloc(22, 'TFI6_N1', tempnn(6), 2)
74        setvar = ualloc(23, 'TFI6_N2', tempnn(6), 2)
75        setvar = ualloc(24, 'TFI6_N3', tempnn(6), 2)
76        setvar = ualloc(25, 'TFI7_N1', tempnn(7), 2)
77        setvar = ualloc(26, 'TFI7_N2', tempnn(7), 2)
78        setvar = ualloc(27, 'TFI7_N3', tempnn(7), 2)
79        setvar = ualloc(28, 'TFI8_N1', tempnn(8), 2)
80        setvar = ualloc(29, 'TFI8_N2', tempnn(8), 2)
81        setvar = ualloc(30, 'TFI8_N3', tempnn(8), 2)
82        setvar = ualloc(31, 'TFI9_N1', tempnn(9), 2)
83        setvar = ualloc(32, 'TFI9_N2', tempnn(9), 2)
84        setvar = ualloc(33, 'TFI9_N3', tempnn(9), 2)
85        setvar = ualloc(34, 'TFI10_N1', tempnn(10), 2)
86        setvar = ualloc(35, 'TFI10_N2', tempnn(10), 2)
87        setvar = ualloc(36, 'TFI10_N3', tempnn(10), 2)
88        setvar = ualloc(37, 'TFI11_N1', tempnn(11), 2)
89        setvar = ualloc(38, 'TFI11_N2', tempnn(11), 2)
90        setvar = ualloc(39, 'TFI11_N3', tempnn(11), 2)
91        setvar = ualloc(40, 'TFI12_N1', tempnn(12), 2)
92        setvar = ualloc(41, 'TFI12_N2', tempnn(12), 2)
93        setvar = ualloc(42, 'TFI12_N3', tempnn(12), 2)
94
95        return
96        end
97
98        subroutine cleartfiarray()
99
100   c     clears all TFI arrays
101
102       integer tempnn(12)
103       logical   setvar,ualloc
104
105       setvar = ualloc(43, 'A1', 0, 2)
106       setvar = ualloc(44, 'B1', 0, 2)
107       setvar = ualloc(45, 'C1', 0, 2)
108       setvar = ualloc(46, 'D1', 0, 2)
109       setvar = ualloc(47, 'A2', 0, 2)
```

```
110         setvar = ualloc(48, 'B2', 0, 2)
111         setvar = ualloc(49, 'C2', 0, 2)
112         setvar = ualloc(50, 'D2', 0, 2)
113         setvar = ualloc(51, 'A3', 0, 2)
114         setvar = ualloc(52, 'B3', 0, 2)
115         setvar = ualloc(53, 'C3', 0, 2)
116         setvar = ualloc(54, 'D3', 0, 2)
117         setvar = ualloc(55, 'A4', 0, 2)
118         setvar = ualloc(56, 'B4', 0, 2)
119         setvar = ualloc(57, 'C4', 0, 2)
120         setvar = ualloc(58, 'D4', 0, 2)
121         setvar = ualloc(59, 'A5', 0, 2)
122         setvar = ualloc(60, 'B5', 0, 2)
123         setvar = ualloc(61, 'C5', 0, 2)
124         setvar = ualloc(62, 'D5', 0, 2)
125         setvar = ualloc(63, 'A6', 0, 2)
126         setvar = ualloc(64, 'B6', 0, 2)
127         setvar = ualloc(65, 'C6', 0, 2)
128         setvar = ualloc(66, 'D6', 0, 2)
129         setvar = ualloc(67, 'A7', 0, 2)
130         setvar = ualloc(68, 'B7', 0, 2)
131         setvar = ualloc(69, 'C7', 0, 2)
132         setvar = ualloc(70, 'D7', 0, 2)
133         setvar = ualloc(71, 'A8', 0, 2)
134         setvar = ualloc(72, 'B8', 0, 2)
135         setvar = ualloc(73, 'C8', 0, 2)
136         setvar = ualloc(74, 'D8', 0, 2)
137         setvar = ualloc(75, 'A9', 0, 2)
138         setvar = ualloc(76, 'B9', 0, 2)
139         setvar = ualloc(77, 'C9', 0, 2)
140         setvar = ualloc(78, 'D9', 0, 2)
141         setvar = ualloc(79, 'A10', 0, 2)
142         setvar = ualloc(80, 'B10', 0, 2)
143         setvar = ualloc(81, 'C10', 0, 2)
144         setvar = ualloc(82, 'D10', 0, 2)
145         setvar = ualloc(83, 'A11', 0, 2)
146         setvar = ualloc(84, 'B11', 0, 2)
147         setvar = ualloc(85, 'C11', 0, 2)
148         setvar = ualloc(86, 'D11', 0, 2)
149         setvar = ualloc(87, 'A12', 0, 2)
150         setvar = ualloc(88, 'B12', 0, 2)
151         setvar = ualloc(89, 'C12', 0, 2)
152         setvar = ualloc(90, 'D12', 0, 2)
153
154         setvar = ualloc(7, 'TFI1_N1', 0, 2)
155         setvar = ualloc(8, 'TFI1_N2', 0, 2)
156         setvar = ualloc(9, 'TFI1_N3', 0, 2)
157         setvar = ualloc(10, 'TFI2_N1', 0, 2)
158         setvar = ualloc(11, 'TFI2_N2', 0, 2)
159         setvar = ualloc(12, 'TFI2_N3', 0, 2)
160         setvar = ualloc(13, 'TFI3_N1', 0, 2)
161         setvar = ualloc(14, 'TFI3_N2', 0, 2)
162         setvar = ualloc(15, 'TFI3_N3', 0, 2)
163         setvar = ualloc(16, 'TFI4_N1', 0, 2)
164         setvar = ualloc(17, 'TFI4_N2', 0, 2)
165         setvar = ualloc(18, 'TFI4_N3', 0, 2)
166         setvar = ualloc(19, 'TFI5_N1', 0, 2)
167         setvar = ualloc(20, 'TFI5_N2', 0, 2)
```

```
168       setvar = ualloc(21, 'TFI5_N3', 0, 2)
169       setvar = ualloc(22, 'TFI6_N1', 0, 2)
170       setvar = ualloc(23, 'TFI6_N2', 0, 2)
171       setvar = ualloc(24, 'TFI6_N3', 0, 2)
172       setvar = ualloc(25, 'TFI7_N1', 0, 2)
173       setvar = ualloc(26, 'TFI7_N2', 0, 2)
174       setvar = ualloc(27, 'TFI7_N3', 0, 2)
175       setvar = ualloc(28, 'TFI8_N1', 0, 2)
176       setvar = ualloc(29, 'TFI8_N2', 0, 2)
177       setvar = ualloc(30, 'TFI8_N3', 0, 2)
178       setvar = ualloc(31, 'TFI9_N1', 0, 2)
179       setvar = ualloc(32, 'TFI9_N2', 0, 2)
180       setvar = ualloc(33, 'TFI9_N3', 0, 2)
181       setvar = ualloc(34, 'TFI10_N1', 0, 2)
182       setvar = ualloc(35, 'TFI10_N2', 0, 2)
183       setvar = ualloc(36, 'TFI10_N3', 0, 2)
184       setvar = ualloc(37, 'TFI11_N1', 0, 2)
185       setvar = ualloc(38, 'TFI11_N2', 0, 2)
186       setvar = ualloc(39, 'TFI11_N3', 0, 2)
187       setvar = ualloc(40, 'TFI12_N1', 0, 2)
188       setvar = ualloc(41, 'TFI12_N2', 0, 2)
189       setvar = ualloc(42, 'TFI12_N3', 0, 2)
190
191       return
192       end
```

Listing L.1: alloctfiarray.F

M Sourcecode of definespline

Source code of the functions for conditioning the user defined parameters from the inputfile into a spline defintion.

```
      subroutine defineSpline(xx,yy,zz,initbend,finalbend,n,
     $     surfacenumber,A, B, C, D)

c      * * F E A P * * A Finite Element Analysis Program

c....  Copyright (c) 1984-2004: Regents of the University of California
c                               All rights reserved

c-----[--.----+----.----+----.-----------------------------------------]
c      Purpose: Define an coupling BC for rectangular surfaces.
c      Inputs:
c         ix            - the element connection array in feap
c         x             - the coordinates array in feap
c         x,y,z         - The coordinates for the spline (support points)
c         initbend      - inital bending for the spline
c         finalbend     - final bending for the spline
c         n             - number of support points
c         surfacenumber - indicates which surface gets transformed
c
c      Outputs:
c         A, B, C, D    - Matrices for Splinedefintions
c-----[--.----+----.----+----.-----------------------------------------]

      implicit  none
      integer n, i, j, dir1, dir2, ntdma, nn, offset, surfacenumber
      real*8 A(n-1), B(n-1), C(n-1), D(n-1), S(n), R
      real*8 k(n-2,n-2), xx(n), yy(n), zz(n), h(n-1)
      real*8 x(n), y(n), z(n)
      real*8 initbend, finalbend

      do i = 1, n-1
         do j = 1, n-1
            K(i,j) = 0
         enddo
      enddo

c     Organizing the data
      if (surfacenumber .eq. 1) then
         do i  = 1, n
            x(i) = zz(i)
            y(i) = xx(i)
         enddo
      elseif (surfacenumber .eq. 2) then
         do i  = 1, n
            x(i) = zz(i)
            y(i) = yy(i)
         enddo
      elseif (surfacenumber .eq. 3) then
         do i  = 1, n
            x(i) = yy(i)
            y(i) = zz(i)
```

```
52          enddo
53       elseif (surfacenumber .eq. 4) then
54          do i  = 1, n
55             x(i) = zz(i)
56             y(i) = xx(i)
57          enddo
58       elseif (surfacenumber .eq. 5) then
59          do i  = 1, n
60             x(i) = zz(i)
61             y(i) = yy(i)
62          enddo
63       elseif (surfacenumber .eq. 6) then
64          do i  = 1, n
65             x(i) = yy(i)
66             y(i) = zz(i)
67          enddo
68       endif
69
70 c     Determine the width of the ith interval
71       do i = 1, n-1
72          h(i) = x(i+1) - x(i)
73       enddo
74
75 c     Elements of the tri-diagonal-matrix
76       do i = 2, n-1
77          j = i-1
78          D(j) = 2*(h(i-1) + h(i))
79          A(j) = h(i)
80          B(j) = h(i-1)
81       enddo
82
83 c     Right Side
84       do i = 2, n-1
85          j = i-1
86          C(j) = 6*((y(i+1) - y(i))/h(i) - (y(i)-y(i-1)) / h(i-1))
87       enddo
88
89 c     Solving the tri-diagonal-matrix
90       ntdma = n-2
91       do i = 2, ntdma
92          R = B(i) / D(i-1)
93          D(i) = D(i) - R*A(i-1)
94          C(i) = C(i) - R*C(i-1)
95       enddo
96
97       C(ntdma) = C(ntdma)/D(ntdma)
98
99       i = ntdma-1
100  111  continue
101      if (i .ge. 1) then
102         C(i) = (C(i) - A(i)*C(i+1)) / D(i)
103         i = i-1
104         goto 111
105      endif
106
107      do i = 2, n-1
108         j = i-1
109         S(i) = C(j)
```

```
110         enddo
111
112         S(1) = initbend
113         S(n) = finalbend
114
115   c     Computing the paramters
116         do i = 1, n-1
117            A(i) = (S(i+1) -S(i))/ (6*h(i))
118            B(i) = S(i)/2
119            C(i) = (y(i+1) - y(i)) / h(i) - (2*h(i)*S(i) + h(i)*S(i+1))/6
120            D(i) = y(i)
121         enddo
122
123         return
124         end
```

Listing M.1: definespline.F